VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Automatická konfigurace a monitorování síťových zařízení

Automated Configuration and Network Devices Monitoring

2021 Tomáš Macák

Rád bych na tomto místě poděkoval všem, kteří mi pomohli s touto diplomovou prací a hlavně vedoucímu mé diplomové práce Ing. Pavlu Nevludovi za jeho rady a připomínky.

Abstrakt

Cílem diplomové práce je navrhnout a ověřit možnosti automatizované konfigurace a monitorování síťových zařízení. V teoretické části je provedena rešerše automatizované konfigurace síťových zařízení. V této části jsou popsány datové formáty, protokoly, podpůrné technologie a automatizační nástroje, které jsou používany v rámci procesu automatizované správy sítě. Další část je věnována možnostem monitorování síťových zařízení. V diplomové práci jsou popsány možnosti monitorování síťových zařízení, přičemž je kladen důraz na popis technologií a nástrojů, které jsou použité i v praktické části diplomové práce. V praktické části jsou potom popsané a otestované dva vytvořené automatizované systémy pro konfiguraci a monitorování síťových zařízení včetně vizualizace dat. První projekt je vytvořen pomocí automatizačního nástroje Ansible. Druhý projekt používá automatizační framework Nornir. Pro uložení dat je použita databáze časových řad InfluxDB. Vizualizace dat je provedena v open source nástroji Grafana. Nakonec jsou oba projekty porovnány a zhodnoceny.

Klíčová slova: Ansible, automatizovaná konfigurace, Grafana, InfluxDB, monitorování, Nornir, síťová automatizace, správa konfigurace

Abstract

The main goal of my Master's thesis is to introduce ways to automate configuration and monitoring of network devices. In the theoretical part is done research on automated configuration of network devices. In this part are described data formats, configuration protocols, supporting technologies and automation tools, which are used in network automation. Next part of my thesis introduces ways to monitor network devices. Mainly are described technologies and tools, which are also used in practical part of Master's thesis. In the practical part are described and tested two created automation systems for configuration and monitoring of network devices including data visualization. First project is created in automation tool Ansible. The second one uses automation framework Nornir. A time series database InfluxDB is used for storing data. Data visualization is done in open source tool called Grafana. Both projects are in the end compared and evaluated.

Keywords: Ansible, automated configuration, configuration management, Grafana, InfluxDB, monitoring, network automation, Nornir

Obsah

Seznam použitých zkratek a symbolů

AAA	–	Authentication, Authorization and Accounting
ACL	–	Access-control list
API	–	Application Programming Interface
BGP	–	Border Gateway Protocol
CGI	–	Common Gateway Interface
Cisco IOS	–	Cisco Internetwork Operating System
CLI	–	Command-line interface
CSR	–	Certificate Signing Request
DSL	–	Domain-Specific Language
EIGRP	–	Enhanced Interior Gateway Routing Protocol
EUI-64	–	Extended Unique Identifier 64
FTP	–	File Transfer Protocol
FTPS	–	File Transfer Protocol Secure
GNS3	–	Graphical Network Simulator
GUI	–	Graphic User Interface
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
HTTPS	–	Hypertext Transfer Protocol Secure
IDE	–	Integrated Development Environment
IETF	–	Internet Engineering Task Force
IoT	–	Internet of Things
IP	–	Internet Protocol
IPv4	–	Internet Protocol version 4
IPv6	–	Internet Protocol version 6
JSON	–	JavaScript Object Notation
Junos OS	–	Junos operating system
MIB	–	Management Information Base
MPLS	–	Multiprotocol Label Switching
NAPALM	–	Network Automation and Programmability Abstraction Layer with Multivendor support
NAT	–	Network address translation
NDOUTILS	–	Nagios Data Output Utilities
NETCONF	–	Network Configuration Protocol
NoSQL	–	Not Only SQL
NTP	–	Network Time Protocol
OID	–	Object Identifier

OSPF	–	Open Shortest Path First
OSPFv2	–	Open Shortest Path First version 2
OSPFv3	–	Open Shortest Path First version 3
PIP	–	PIP Installs Packages
QEMU	–	Quick EMUlator
RAM	–	Random-access memory
REST	–	Representational state transfer
RESTCONF	–	REST Configuration Protocol
RFC	–	Request for Comments
RPC	–	Remote Procedure Call
SCP	–	Secure copy protocol
SGML	–	Standard Generalized Markup Language
SMTP	–	Simple Mail Transfer Protocol
SNMP	–	Simple Network Management Protocol
SOAP	–	Simple Object Access Protocol
SQL	–	Structured Query Language
SSH	–	Secure Shell
SSL	–	Secure Sockets Layer
SVI	–	Switch Virtual Interface
SVN	–	Subversion
TCP	–	Transmission Control Protocol
TIG	–	Telegraf, InfluxDB, Grafana
TLS	–	Transport Layer Security
TSD	–	Time Series Daemon
TSDB	–	Time Series Database
UDP	–	User Datagram Protocol
UML	–	Unified Modeling Language
URL	–	Uniform Resource Locator
UTC	–	Coordinated Universal Time
UTF-8	–	Universal Transformation Format-8
VLAN	–	Virtual LAN
VTP	–	VLAN Trunking Protocol
W3C	–	The World Wide Web Consortium
XML	–	Extensible Markup Language
XPath	–	XML Path Language
XSD	–	XML Schema Definition
XSLT	–	Extensible Stylesheet Language Transformations
XQuery	–	XML Query
YAML	–	YAML Ain't Markup Language

YANG – Yet Another Next Generation

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

Úvod

Se zvyšujícím se počtem požadavků na dostupnost a kvalitu poskytovaných služeb jsou kladeny stále vyšší nároky na síťové infrastruktury. Správa komplexních síťových infrastruktur musela být v minulosti prováděna tak, že se správce sítě musel manuálně připojit pomocí komunikačního protokolu SSH (Secure Shell) k jednotlivým síťovým zařízením a pomocí CLI (Command-line interface) musel zadat sérií příkazů pro konfigurování každého zařízení zvlášť. Manuální přístup ke konfiguraci síťových zařízení není velmi efektivní. Se zvyšující se popularitou automatizace úkolů vzešel proces tzv. network automation (síťové automatizace), který se podílí na automatizované konfiguraci, testování a správě síťových zařízení pomocí softwarových prostředků. Monitorováním síťových zařízení lze získat přehled o stavu jednotlivých zařízení, což je velmi důležité při diagnostice případných chyb v počítačové síti. Cílem diplomové práce je navrhnout a ověřit možnosti automatizované konfigurace a monitorování síťových zařízení.

Teoretická část je rozdělena do dvou kapitol. První kapitola se věnuje problematice automatizace síťových konfigurací a hromadné správy síťových zařízení a serverů. Nejdříve je provedena rešerše různých přístupů k automatizované správě síťových zařízení včetně druhů rozhraní, která zařízení využívají pro automatizaci. Dále jsou popsány různé strukturované datové formáty, které se používají pro serializaci dat a tvorbu inventářů. Dále je provedena rešerše různých typů Web Service API. Z Web Service APIs potom vycházejí standardizované protokoly pro správu síťových zařízení. V práci je provedena rešerše používaných modelovacích jazyků s důrazem na modelovací jazyk YANG (Yet Another Next Generation). Dále jsou popsány podpůrné technologie a automatizační nástroje, které lze využít pro automatizovánou správu síťových zařízení a serverů. V poslední části první kapitoly byl proveden stručný popis verzovacích systémů včetně jejich použití. Druhá kapitola se věnuje monitorováním síťových zařízení. Nejprve je popsán protokol SNMP (Simple Network Management Protocol) včetně jeho nedostatků, které jsou adresovány principem Streaming telemetry. Důraz je v této kapitole kladen na popis moderního přístupu k monitorování síťových zařízení a serverů téměř v reálném čase. V rámci práce byly popsány možnosti sběru dat časových řad, jejich uložení a následnou vizualizaci.

Praktická část diplomové práce se věnuje návrhu a ověření automatizovaného systému pro konfiguraci a monitorování síťových zařízení a serverů. Praktická část je rozdělna do čtyř kapitol. První kapitola je věnována analýze automatizovaného systému včetně výběru použitých technologií a nástrojů pro implementaci. Dále je zde uvedena použitá síťová topologie a přehled testovaných obrazů emulovaných zařízení. Druhá kapitola je věnována návrhu a ověření automatizovaného systému, který je realizován nástroji Ansible a TIG (Telegraf, InfluxDB, Grafana) stack. Třetí kapitola je věnována návrhu a ověření automatizovaného systému, který je realizován nástroji Nornir a TIG stack. V poslední kapitole jsou obě implementace automatizovaného systému zhodnoceny a porovnány.

1 Automatizace síťových konfigurací

Automatizace síťových konfigurací je jednou z oblastí, která spadá do procesu network automation. Automatizováním síťových konfigurací lze dynamicky vytvářet konfigurace jednotlivých zařízení (dle požadavků správce sítě) a tyto konfigurace aplikovat paralelně na předem definovaná zařízení. Jednotlivá zařízení, včetně jejich parametrů, jsou definována v specifických datových souborech neboli inventářích.

1.1 Imperativní přístup ke konfiguraci síťových zařízení

Při konfiguraci síťového zařízení probíhá komunikace mezi uživatelem/aplikací a síťovým zařízením za použití komunikačního protokolu SSH. Rozhraní příkazové řádky (CLI) je použito pro konfiguraci síťového zařízení. Do CLI uživatel/aplikace přímo zadá konkrétní příkazy, které postupně síťové zařízení vykoná. V praxi se lze setkat s tzv. imperativním přístupem ke konfiguraci síťových zařízení. Uživatel musí pomocí příkazů definovat jednotlivé kroky konfigurace, tak aby docílil požadované funkcionality. [1]

Nevýhody imperativního přístupu:

1. Nutnost definovat jednotlivé kroky konfigurace.
2. Odklon od samotného problému, který správce řeší v dané chvíli.
3. Příkazy nejsou sjednoceny mezi jednotlivými výrobci (vendory).
4. Nepodporována idempotence. Před nahráním nové konfigurace není kontrolován současný stav konfigurace. Všechny příkazy jsou provedeny. Ve výsledku může dojít k nechtěné změně v konfiguraci zařízení.

```
R1#
R1#enable
R1#configure t
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#interface fastEthernet 0/0
R1(config-if)#description popis
R1(config-if)#ip address 192.168.70.1 255.255.255.0
R1(config-if)#no shu
R1(config-if)#no shutdown
R1(config-if)#exit
R1(config)#
```

Obrázek 1: Imperativní přístup ke konfigurace síťového rozhraní na zařízení Cisco c7200

1.2 Deklarativní přístup ke konfiguraci síťových zařízení

Deklarativní přístup k síťové konfiguraci (tzv. intent-based networking) adresuje problémy imperativního přístupu. Důraz je tedy kladen na řešení samotného problému. Ve srovnání s im-

perativním přístupem není nutné definovat jednotlivé kroky konfigurace. Deklarativní přístup umožňuje uživateli definovat přímo stav, ve kterém se síťové zařízení má nacházet. Pokud tedy konfigurujeme např. síťové rozhraní daného zařízení, tak nezadáváme jednotlivé příkazy do CLI, ale deklarujeme konečný stav síťového rozhraní za pomocí knihoven třetích stran nebo implementovaného API (Application Programming Interface). [1]

Deklarativní přístup podporuje idempotenci. Před přidáním nové konfigurace je porovnán současný stav síťového zařízení se stavem po aplikování nové konfigurace. Pokud jsou stavy mezi sebou ekvivalentní, tak nová konfigurace není aplikována. Pokud jsou stavy navzájem rozdílné, tak jsou do síťového zařízení aplikovány pouze změny z nové konfigurace. [3]

Nevýhody deklarativního přístupu:

1. Nutnost se naučit s novými technologiemi v oblasti automatizace (datové formáty, protokoly, API, podpůrné knihovny vytvořené komunitou, automatizační nástroje).
2. U starších síťových zařízení, u kterých lze využít pouze CLI, se lze spoléhat pouze na knihovny, které jsou vytvořené komunitou. Tyto knihovny nejsou podporovány všemi síťovými zařízeními.

```
vvvv IPv4 interfaces config ** changed : False vvvvvvvvvvvvvvvvvvvvvvvv
---- IPv4 Interfaces Configuration ** changed : False ------------------
interface FastEthernet0/0
 description popis
 ip address 192.168.70.1 255.255.255.0
 duplex full
 speed auto
 no shutdown
!
end
```

Obrázek 2: Deklarativní přístup ke konfigurace síťového rozhraní na zařízení Cisco c7200

1.3 Datové formáty

Datové formáty slouží k reprezentaci informací. Síťová zařízení, které poskytují pouze CLI, umí přehledně zobrazit uživateli výsledky zadaných příkazů. Pokud uživatel chce s daty dále pracovat (např. pomocí skriptu ukládat data do databáze), tak je musí nejdříve pracně zparsovat pomocí regulárních výrazů. Příčinou je, že uživatel obdrží data v nestrukturovaném formátu (textový řetězec).

Vývojem webových API se změnil i způsob, jakým jsou data reprezentována pro jejich přenos. V oblasti network automation umožňuje vhodná reprezentace dat efektivnější výměnu dat mezi API jednotlivých zařízení a uživatelskými aplikacemi. Data jsou reprezentována ve strukturovaných formátech, které výrazně usnadňují parsování chtěných informací. Různé implementace Web service APIs (viz oddíl č. 1.4) reprezentují data ve strukturovaném formátu XML (Extensible Markup Language) nebo JSON (JavaScript Object Notation). V rámci network automation má významnou roli strukturovaný formát YAML (YAML Ain't Markup Language), který se využívá pro vytváření statických inventářů.

1.3.1 YAML

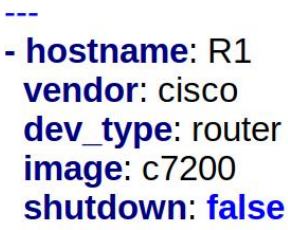
Strukturovaný formát určený pro serializaci dat, který je však uzpůsoben tak, aby byl čitelný i člověkem. Čitelnost tohoto formátu je hlavním důvodem, proč se používá v automatizačních nástrojích jako datový formát pro tvorbu statických inventářů. YAML formát lze jednoduše zparsovat pomocí volně dostupných knihoven (např. v jazyce Python lze využít knihovnu PyYAML). Datový formát je momentálně ve verzi 1.2. Ve srovnání s předchozími verzemi je tato verze zaměřena na kompatabilitu JSON formátu s YAML formátem. Dle YAML specifikace v1.2 je JSON formát podmnožinou formátu YAML. Jakákoliv validní reprezentace dat v JSON formátu je validní reprezentací dat v YAML formátu. [2, 3]

Podporované datové typy: [2]

- **Skalární (string, int, float, boolean)**
- **List (sekvence)** - seřazený seznam/pole hodnot
- **Slovník (mapy)** - asociativní pole

Základní charakteristika formátu:

- Datový soubor s příponou .yaml nebo .yml.
- Na začátku souboru jsou vždy tři pomlčky (- - -).
- Informace je vždy zapsána jako pár klíč:hodnota (viz obrázek č. 3).
- Pro tvorbu hierarchie komplexních struktur je důležitá indentace. Dle specifikace YAML v1.2 se pro předsazení využívají mezery, ale jejich počet není přesně specifikován. Editory a IDE (Integrated Development Environment) většinou v defaultní konfiguraci používají pro předsazení na další úroveň 2 mezery. [2]



Obrázek 3: Ukázka YAML souboru - popis Cisco směrovače

Mezi největší výhody YAML formátu patří jeho čitelnost. Dále formát nabyl v posledních letech velké podpory (komunitou podporované knihovny v různých programovacích jazycích, oblíbený datový formát u některých automatizačních nástrojů). Pro uživatele je formát velmi jednoduchý na používání a naučení (intuitivní syntax). Největší nevýhodou formátu je nutnost přesné indentace při používání komplexních struktur. Na druhou stranu při použití IDE je indentace automaticky kontrolována. [4]

YAML formát má rozsáhlé využití. Obecně se využívá pro serializaci dat. V oblasti network automation se nejčastěji používá pro tvorbu statických inventářů (např. u automatizačních nástrojů Ansible, Nornir) a vytváření Ansible scénářů.

1.3.2 JSON

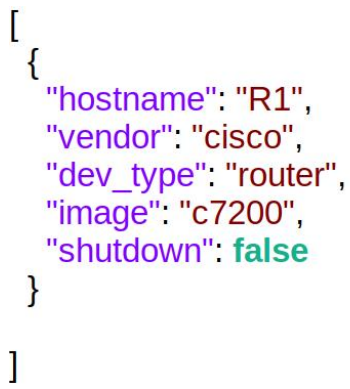
Strukturovaný formát, který je určen pro serializaci dat. Textový formát, který přejímá datové typy používané v programovacím jazyce JavaScript (definované standardem ECMA-262 a ECMA-404). JSON tvoří podmnožinu formátu YAML. Specifikace formátu a jeho syntax jsou publikované v dokumentech IETF (Internet Engineering Task Force) RFC (Request for Comments) 8259, ECMA-404 a ECMA-262 Fifth Edition. [5] Informace v JSON formátu je zaznamenána jako pár klič:hodnota (viz obrázek č. 4). JSON umožňuje vytvářet komplexní datové struktury (např. vnořené objekty, vícerozměrné listy).

Na rozdíl od YAML formátu zde není kladen důraz na čitelnost člověkem, ale především na jednoduchost a celkově efektivní přenos dat mezi zařízeními (parsování, podpora v moderních programovacích jazycích). Nevýhodou tohoto formátu je v některých případech jeho nerobustnost. V takovém případě používají vývojáři formát XML, který disponuje řadou podpůrných technologií a jazyků jako například XSD (XML Schema Definition), XLST (Extensible Stylesheet Language Transformations), XPath (XML Path Language) a XQuery (XML Query).

Podporované datové typy: [6]

- **string** - sekvence Unicode znaků, jako defaultní kódování použito UTF-8 (Universal Transformation Format-8)

- **number** - celá a desetinná čísla (standard IEEE 754-2008 binary64)
- **boolean** - hodnota true nebo false
- **null** - reprezentace chybějící hodnoty
- **object** - asociativní pole, podobné slovníku (dictionary), páry klíč:hodnota oddělené čárkou
- **array** - seřazená sekvence hodnot (pole)



Obrázek 4: Ukázka JSON souboru - popis Cisco směrovače

1.3.3 XML

Strukturovaný textový formát, který se především využívá pro serializaci dat. Formát je založen na standardu SGML (Standard Generalized Markup Language). Ve srovnání s ostatními formáty je XML velmi robustní. U XML dokumentů (viz obrázek č. 5) se používají dodatečné technologie a jazyky (např. XSD, XSLT, XPath, XQuery), které umožňují XML dokumenty validovat, efektivně vyhledávat potřebné data nebo transformovat XML dokument do jiného formátu [3]. V posledních letech upadá podpora tohoto formátu ve prospěch datového formátu JSON.

Struktura XML dokumentu je složena z následujících částí [7, 8]:

- **XML prologem** - XML prolog je tvořen XML deklarací, která obsahuje XML verzi a použité kódování znaků v XML dokumentu.
- **Jmennými prostory (nepovinné)** - Řeší konflikty názvů elementů a atributů (v případě kolizí).
- **XSD schéma (nepovinné)** - Reference k použitému XSD schématu, podle kterého je XML dokument validován.

- **Kořenový element** - Hlavní element, který obaluje další elementy. Nutno definovat vlastní začínající značku a ukončovací značku.
- **Elementy** - V kořenovém elementu, tvořeny začínajícím tagem, atributy, hodnotami a ukončovacím tagem. Hodnotou může být konkrétní hodnota nebo další element.

Výhodou XML formátu je především jeho intuitivní syntax, flexibilita a možnost validace XML dokumentu pomocí XSD schémat. Nevýhodou formátu je redundance v syntaxi, méně efektivní přenos (větší množství dat na přenos kvůli redundanci), složitější na parsování a sběru dat (na rozdíl od JSON formátu). XML formát nemá definované datové typy na rozdíl od JSON nebo YAML formátu (popis XML elementů včetně jejich datových typů lze provést definováním XSD schématu).

Použití datového formátu [9]:

- Vytváření HTML (Hypertext Markup Language) dokumentů pomocí jazyka XSLT.
- V implementacích SOAP (Simple Object Access Protocol) API a některých implementacích REST (Representational state transfer) API.

```
<?xml version="1.0" encoding="UTF-8"?>
<devices>
    <device>
        <hostname>R1</hostname>
        <vendor>cisco</vendor>
        <dev_type>router</dev_type>
        <image>c7200</image>
        <shutdown>false</shutdown>
    </device>
</devices>
```

Obrázek 5: Ukázka XML souboru - popis Cisco směrovače

1.3.4 Porovnání

-	YAML	JSON	XML
Ukládání dat	klíč:hodnota	klíč:hodnota	elementy
Výhody	čitelnost, jednoduchost	jednoduché parsování, podpora	robustnost, podpůrné technologie
Nevýhody	indentace	-	redundance, robustnost
Použití	inventáře, Ansible scénáře	implementace REST API	implementace SOAP API, HTML dokumenty

Tabulka 1: Porovnání používaných datových formátů v network automation

1.4 Web Service APIs

S vývojem technologií pro vývoj webových aplikací, mikroslužeb a cloudových technologií vzešla nutnost efektivně přenášet strukturované data po počítačové síti. Na základě těchto požadavků vznikly různé protokoly a pravidla, která lze využít při implementaci Web Service API. Web Service API se využívá především pro výměnu strukturovaných dat mezi klientskou aplikací a serverem nebo mezi jednotlivými mikroslužbami.

1.4.1 XML-RPC

XML-RPC je RPC (Remote Procedure Call) protokol, který lze použít pro přenos dat mezi systémy. Protokol vznikl v roce 1998. XML-RPC využívá aplikační protokol HTTP (Hypertext Transfer Protocol) nebo HTTPS (Hypertext Transfer Protocol Secure) pro komunikaci mezi XML-RPC klientem a serverem. Jednotlivé RPC zprávy (včetně odpovědí) jsou reprezentovány ve formátu XML. Pravidla a forma XML-RPC zpráv jsou popsána v samotné specifikaci protokolu. [10, 11]

1.4.2 JSON-RPC

JSON-RPC je RPC protokol, který lze použít pro přenos dat mezi systémy. Protokol vznikl v roce 2005. Pro interakci mezi klientem a JSON-RPC serverem se nejčastěji používá aplikační protokol HTTP/HTTPS. Všechny RPC zprávy mezi klientem a serverem jsou reprezentovány ve formátu JSON. [11, 12]

1.4.3 SOAP

SOAP (Simple Object Access Protocol) je standardizovaný protokol organizací W3C (The World Wide Web Consortium), který se využívá pro výměnu strukturovaných dat mezi systémy. Nejnovější specifikací SOAP protokolu je verze 1.2, která vyšla v roce 2007. Pro komunikaci mezi klientem a serverem lze využít například aplikační protokoly HTTP/HTTPS, SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol). Primárně je využíván protokol HTTP/HTTPS. Jednotlivé SOAP zprávy jsou reprezentovány datovým formátem XML. SOAP protokol nepodporuje datový formát JSON. Struktura SOAP zpráv je jasně definována ve specifikaci protokolu [14]. SOAP má řadu rozšíření, které lze například použít pro šifrování zpráv, výměnu metadat nebo správu transakcí. [14, 15]

1.4.4 REST

REST (Representational state transfer) je architektura rozhraní, jenž je definována sadou pravidel [16]. REST architektura adresuje problémy SOAP protokolu, který neumožňuje reprezentovat zprávy v JSON formátu. Mimo jiné samotná struktura SOAP zpráv je velmi rigidní. Dnešní implementace REST API využívají zejména protokol HTTP/HTTPS pro bezestavovou komunikaci

mezi klientem a serverem. HTTPS poskytuje metody pro samotnou komunikaci, zabezpečení komunikace a cachování. Žádané data lze v těle HTTP zprávy reprezentovat v různých datových formátech (JSON, XML, HTML atd.). Nejvíce využívaný je datový formát JSON. [17, 18]

1.4.5 gRPC

gRPC je open source RPC framework, který je vyvíjený společností Google. Prvotní verze frameworku byla vydána v roce 2016. Implementace frameworku je založena na protokolu HTTP/2, který je použit pro komunikaci mezi serverem a klientem. gRPC provádí serializaci dat pomocí technologie Protocol Buffers. Pomocí technologie Protocol Buffers lze popsat strukturu definovaných dat. Popsaná datová struktura je následně zkompilována Protobuf kompilátorem. Protobuf zpráva, reprezentována v binárně podobě, je následně zaslána serveru/klientovi. [19]

1.5 Protokoly pro správu síťových zařízení

Se zvyšující se popularitou Web Service APIs a automatizaci procesů byly vyhledávány možnosti, jak využít tohoto vývoje ve prospěch automatizované správy a sběru dat ze síťových zařízení. Na základě toho vznikly dva protokoly NETCONF (Network Configuration Protocol) a RESTCONF (REST Configuration Protocol), které umožňují zpřístupnit a komunikovat s implementovaným API síťového zařízení. API síťového zařízení lze například využít pro interakci s klientskou aplikací (například pro automatickou správu zařízení, rollback konfigurace, sběr dat atd.). Mimo jiné protokoly adresují problémy při automatizováné správě síťových zařízení poskytující pouze CLI.

Hlavní problémy při používání CLI: [3]

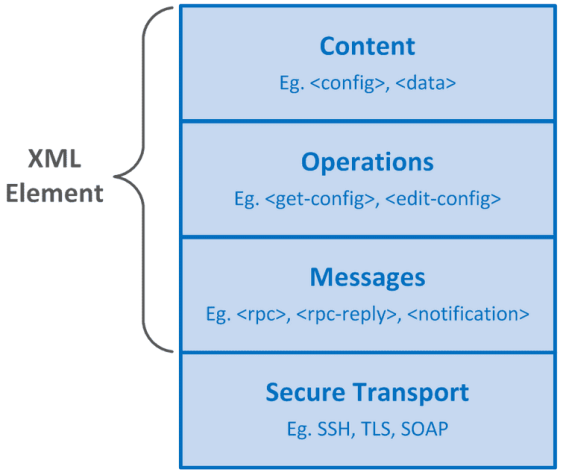
- CLI je navržen pro člověka. CLI vrací nestrukturované data (pouze textový řetězec). Informaci je nutné manuálně zparsovat regulárními výrazy.
- Není podporována idempotence při automatizované správě síťového zařízení. Využívá se imperativní přístup k vytváření síťové konfigurace.

1.5.1 NETCONF

NETCONF je protokol standardizovaný organizací IETF, který se používá pro správu síťových zařízení a sběru dat z těchto zařízení. Mimo jiné operace NETCONF protokolu podporují například návrat k předchozí konfiguraci nebo zálohování/obnovení konfigurace. Protokol je specifikován v dokumentech RFC 6241 a RFC 6242. Veškeré NETCONF zprávy jsou reprezentovány datovým formátem XML. NETCONF operace a samotné struktury XML zpráv jsou definovány standardem protokolu. [20, 21]

Popis vrstev protokolu (viz obrázek č. 6): [21, 22]

- **Content** - Vrstva obsahu, která se skládá z konfiguračních nebo stavových dat, která jsou reprezentována v XML formátu. V posledních letech se využívá YANG modelů k vytvoření a validování struktury konfiguračních nebo stavových dat.
- **Operations** - Vrstva operací, která definuje operace, které lze provést na NETCONF serveru (síťovém zařízení).
- **Messages** - Vrstva zpráv, která poskytuje mechanismus pro zakódování RPC zpráv a notifikací.
- **Secure Transport** - Transportní vrstva, která zajišťuje komunikaci mezi NETCONF klientem a serverem. Spojení mezi NETCONF serverem a klientem musí být zabezpečené a persistentní. Nejpoužívanější jsou protokoly SSH nebo TLS (Transport Layer Security).



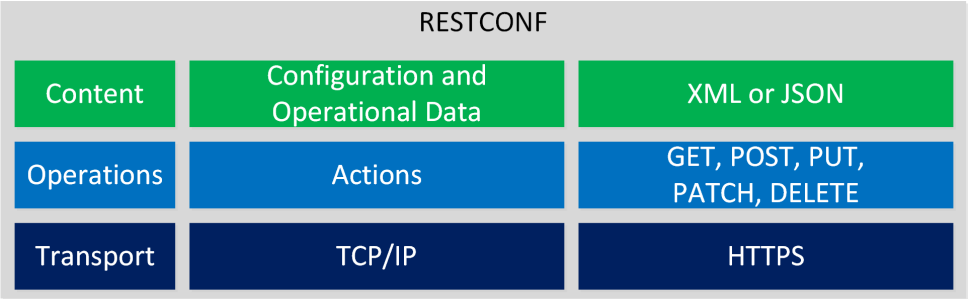
Obrázek 6: Vrstvy protokolu NETCONF [22]

1.5.2 RESTCONF

Vzhledem k obrovské popularitě REST API vznikl standardizovaný protokol RESTCONF (REST Configuration Protocol), jenž představuje jednodušší alternativu k protokolu NETCONF. RESTCONF protokol se řídí zásad REST architektury. Protokol je specifikován v dokumentu RFC 8040. Na rozdíl od NETCONF protokolu nabízí RESTCONF však pouze omezenou funkcionalitu. Na druhou stranu se využívá už zavedených způsobů, které vzešly v rámci specifikace NETCONF protokolu (např. úložiště konfigurace včetně vytváření/validování datových struktur pomocí YANG modelů). Na rozdíl od NETCONF protokolu lze konfigurační a stavová data reprezentovat v datových formátech JSON nebo XML. [20, 23]

Popis vrstev protokolu (viz obrázek č. 7): [23]

- **Content** - Vrstva obsahu, která se skládá z konfiguračních nebo stavových dat, která jsou reprezentována v XML nebo JSON formátu.
- **Operations** - Vrstva operací, která je však ve srovnání s NETCONF protokolem definována přímo HTTP metodami (GET, POST atd.).
- **Secure Transport** - Transportní vrstva, která zajišťuje komunikaci mezi RESTCONF klientem a serverem. Komunikace musí být zabezpečená a dle REST architektury bezstavová. Transportní vrstva využívá protokol HTTPS.



Obrázek 7: Vrstvy protokolu RESTCONF [24]

1.6 Datové modely a schémata

Datové modely a schémata popisují, jak mají být data v datovém formátu strukturována. Datové modely a schémata slouží především k validaci datového souboru dle potřeb uživatele. Pomocí modelovacích jazyků lze definovat konkrétní datové typy, konstrukce a další parametry, které nutně musí mít data konkrétního datového souboru. Datové modely lze vytvářet pomocí modelovacích jazyků jako například YANG (Yet Another Next Generation), XSD, JSON Schema a nebo pomocí knihoven třetích stran. Pro YAML formát není momentálně vyvinut standardizovaný modelovací jazyk, který by umožňoval validovat data YAML souborů. Na druhou stranu existují knihovny (např. Pykwalify), které lze použít pro validování dat v YAML dokumentu. [3]

1.6.1 JSON Schema

JSON Schema je modelovací jazyk, který se používá pro validaci a popis JSON souborů. JSON schémata jsou reprezentováná v JSON formátu s konstrukcemi a klíčovými slovy, které umožňují definovat strukturu a obsah JSON dokumentu. Pomocí JSON Schema lze například definovat datový typ hodnoty a rozmezí, v jakém se hodnota musí nacházet. Dále lze definovat celkovou strukturu dokumentu (názvy klíčů, datové typy, metadata, volitelné/povinné klíče atd.). [25,26]

1.6.2 XSD

XSD patří mezí modelovací jazyky, které slouží k vytváření XSD schémat. Výsledné XSD schéma lze použít ke generování nebo validování XML dokumentů. Modelovací jazyk využívá specifika datového formátu XML a dodatečné syntaxe, která je standardizována organizací W3C. Tato syntax umožňuje uživatelům deklarovat strukturu XML dokumentu (pořadí elementů, atributy, datové typy, rozsah hodnot atd.). [27, 28]

1.6.3 YANG

YANG je modelovací jazyk, který se používá k modelování konfiguračních dat, stavových dat a RPC (Remote Procedure Calls) [29]. Specifikace YANG v1.1 je zdokumentována v dokumentu RFC 7950. Na rozdíl od XSD a JSON Schema je YANG vyvinut především pro potřeby network automation (přesněji automatizaci za využití implementovaných API). Mimo jiné lze pomocí YANG modelů validovat data, která jsou reprezentována v JSON nebo XML formátu. YANG je komplementarní s různými druhy API a protokoly (především s NETCONF a RESTCONF) [3, 30]. Každý výrobce síťových zařízení implementuje vlastní YANG modely, přičemž i v rámci jednoho výrobce mají zařízení rozličnou podporu YANG modelů.

Modelovací jazyk YANG podporuje velké množství datových typů (např. boolean, int32, string, binary, decimal64) [29]. Pokud vestavěné datové typy nesplňují požadavky uživatele, tak je možné nadefinovat vlastní datové typy. Vlastní datové typy musí však vycházet z vestavěných datových typů. Příkladem může být datový typ, který umožňuje validaci IPv4 (Internet Protocol version 4) adres pomocí regulárních výrazů.

Základní struktura YANG modelu (příklad viz obrázek č. 8): [31]

- **Module** - Modul je definován na začátku YANG modelu. Jméno modulu je ekvivalentní názvu souboru.
- **Metadata** - YANG verze, jménné prostory, prefix modulu a informace o samotném modelu (popis, číslo revize, organizace).
- **Odvozené datové typy** - Datové typy, které jsou odvozené od vestavěných datových typů.
- **Datové uzly** - Modelují strukturu dat (konkrétně stromová struktura). V YANG specifikaci v1.1 jsou definovány 4 typy datových uzlů.

Typy datových uzlů: [29, 31]

- **Leaf** - Datový uzel, pomocí kterého lze deklarovat datový typ pouze pro jednu hodnotu. Nemá žádné potomky.

- **Leaf-List** - Sekvence **leaf** uzlů. **Leaf-List** umožňuje vytvořit několik instancí daného **leaf** uzlu.
- **List** - Sekvence **list** záznamů. Každý **list** záznam je speciální typ kontejneru, který seskupuje ostatní datové uzly. Záznamy jsou unikátně rozlišeny pomocí definovaného klíče (**id**). **List** umožňuje vytvářet komplexní stromové struktury.
- **Container** - Kontejnery seskupují související datové uzly. Na rozdíl od ostatních datových uzlů neobsahuje žádnou hodnotu.

```
module devices{ //modul
  yang-version 1.1;
  namespace "dp.macak";
  prefix if;
  container devices { //kontejner devices
    list device { //sekvence device
      key "hostname";
      leaf hostname {
        type string;
      }
      leaf vendor {
        type string;
      }
      leaf dev_type {
        type string;
      }
      leaf image {
        type string;
      }
      leaf shutdown {
        type boolean;
      }
    }
  }
```

Obrázek 8: Ukázka YANG modelu - popis síťového zařízení

1.7 Podpůrné technologie pro automatizaci síťové konfigurace

Automatizace síťové konfigurace není pouze o pochopení rozličných datových formátů, aplikačních rozhraní, protokolů a automatizačních nástrojů. Automatizační nástroje využívají v rámci automatizace síťových konfigurací řadu knihoven nebo modulů, které zjednodušují interakci se zařízeními různých výrobců. K dispozici je velké množství Python knihoven a modulů, které lze využít pro automatizaci konfigurace síťových zařízeních [1, 32]. Nejvíce využívané jsou potom **multi-vendor** knihovny a moduly, které umožňují automatizovat konfigurace síťových zařízení od různých výrobců. V posledních letech se vývoj knihoven/modulů ubírá směrem k multi-vendor knihovnám/modulům, které poskytují jednotné rozhraní k automatizování konfigurace a sběru dat ze síťových zařízení. Příkladem takové knihovny je **NAPALM** (Network Automation and Programmability Abstraction Layer with Multivendor support).

Dalšími podpůrnými nástroji na automatizaci síťových konfigurací jsou šablonovací systémy. Šablonovací systémy hrály vždy významnou roli při generování dynamických webových stránek.

Šablonovací systémy jsou zejména využívány moderními webovými frameworky (Django, Flask, Angular, React, Vue.js atd.). Popularita šablonovacích systémů se projevuje i při procesu automatizování konfigurace síťových zařízení. Pomocí šablonovacích systemů lze vytvářet šablony (templates) jednotlivých síťových konfigurací. Šablona síťové konfigurace je tvořena statickými částmi, které jsou neměnné a dynamickými částmi, jejichž hodnoty jsou později dynamicky přidány dle potřeb uživatele. Vkládání hodnot do dynamických částí šablony je provedeno při renderování šablony. V rámci automatizace síťových konfigurací je nejvíce populární šablonovací systém **Jinja2**, který má velkou podporu u jednotlivých automatizačních nástrojů. [3]

1.7.1 Příklady Python knihoven pro konfiguraci a sběr dat ze síťových zařízeních

Paramiko: Paramiko je Python knihovna, která implementuje protokol SSHv2. Knihovna není uzpůsobena pro automatizaci konfigurace a sběru dat ze síťových zařízení. Paramiko je hlavně využívána jako generická knihovna, která je dále rozšiřována jinými knihovnami dle potřeb vývojářů. [1, 34, 35]

Netmiko: Multi-vendor knihovna, která slouží jako **rozšíření knihovny Paramiko**. Knihovna Netmiko je uzpůsobena pro automatizaci konfigurace a sběru dat ze síťových zařízení, které využívají CLI rozhraní a SSH protokol pro interakci se zařízením. Netmiko knihovna má ve srovnání s Paramiko knihovnou přímo implementované metody, pomocí kterých lze jednoduše posílat příkazy síťovým zařízením a sbírat **nestrukturované data** z těchto zařízení. Na rozdíl od Paramiko knihovny není taky nutné specifikovat počet bajtů, které lze naráz obdržet z SSH kanálu. **Netmiko nekontroluje aktuální stav konfigurace síťového zařízení, pouze posílá uživatelem definované příkazy a vrací nestrukturované data ve formě textového řetězce.** [1, 34, 35]

ncclient: Python knihovna, která implementuje NETCONF klienta pro interakci se síťovým zařízením (přesněji NETCONF serverem) pomocí NETCONF protokolu. [3]

requests: Python knihovna, která usnadňuje zasílání HTTP (Hypertext Transfer Protocol) požadavků. Práce s knihovnou je velmi intuitivní. Hlavní využití má při automatizování konfigurace a sběru dat ze síťových zařízení, které mají implementované REST API. REST API musí být však založeno na HTTP protokolu. Používá se tedy i pro síťové zařízení, které podporují RESTCONF protokol. [3]

junos-eznc: Python knihovna, která zjednodušuje konfiguraci a sběr dat z Juniper zařízeních. Získaná data ze síťového zařízení jsou reprezentována strukturovaně. Knihovna především rozšiřuje možnosti Python knihovny **ncclient**. [1]

NAPALM: Multi-vendor knihovna, která poskytuje uživatelům **uniformní rozhraní pro**

konfiguraci síťových zařízení a sběru dat ze síťových zařízení různých vendorů a operačních systémů [3]. Pokud máme např. dvě síťová zařízení různých vendorů (přesněji operačních systémů), které poskytují jiné rozhraní (první CLI a druhé API), tak NAPALM umožňuje jednotný přístup k obdrženým datům. NAPALM vytváří abstraktní vrstvu nad knihovnami jako např. **netmiko** (pro Cisco zařízení), **junos-eznc** (pro Juniper zařízení). Na základě operačního systému síťového zařízení je nutné inicializovat specifický driver. Driver potom určuje i knihovnu, která se bude používat pro interakci se zařízením [36]. NAPALM umožňuje **deklarativní přístup k síťové konfiguraci** (i pro síťové zařízení využívající CLI). Pomocí NAPALM knihovny lze **nahradit** současnou běžící síťovou konfiguraci za zcela novou konfiguraci (operace **replace**) nebo **sloučit** novou konfiguraci se současně běžící konfigurací (operace **merge**). NAPALM poskytuje metody pro návrat k předcházející konfiguraci (operace rollback) nebo porovnání nové konfigurace vůči současné běžící konfiguraci (operace **configuration compare**). [3, 37]

Nevýhody **NAPALM** knihovny:

- NAPALM podporuje pouze některé výrobce a operační systémy. Seznam podporovaných operačních systémů a výrobců je uveden v dokumentaci NAPALM knihovny [36].
- NAPALM poskytuje pouze omezený počet metod (tzv. getterů) pro sběr dat ze síťových zařízení. Některé getter metody nejsou podporovány pro dané NAPALM ovladače (drivery). Seznam getterů je uveden v dokumentaci NAPALM knihovny [36].
- NAPALM ovladač **ios** (Cisco IOS) využívá pro interakci se síťovým zařízením knihovnu Netmiko [36]. Netmiko provede na síťovém zařízení dané příkazy a vrátí list textových řetězců, které jsou pomocí regulárních výrazů (definovaných v NAPALM knihovně) zparsovány. Uživatel po zparsování obdrží strukturované data. **Pokud by se však v budoucnu změnil výstupní textový řetěz (např. kvůli nové verzi Cisco IOS), tak definované regulární výrazy nemusí správně zpracovat daný textový řetězec. V takovém případě uživatel získá pouze programátorem definované výchozí hodnoty**.

1.7.2 Jinja2

Šablonovací systém, jehož syntaxe je velmi podobná syntaxi programovacího jazyka Python. Jinja2 má rozsáhlé využití v oblasti tvorby webových stránek (např. pomocí webového frameworku Flask) a network automation. V rámci procesu network automation se nejčastěji využívá pro generování dynamických konfiguračních souborů a vytváření různých typů reportů (HTML). Jinja2 šablona (viz obrázek č. 9) je složena ze statických a dynamických částí. Při automatizaci síťových konfigurací tvoří statické části konfigurace neměnnou část konfigurace, která je identická pro síťová zařízení se stejným operačním systémem. Dynamické části šablony jsou potom tvořeny Jinja2 výrazy a referencemi k proměnným, které jsou potom renderovány. [3, 33]

Stručný popis základní Jinja2 syntaxe: [33, 38]

- **Proměnné** - Uživatel může vytvářet nové proměnné a přiřazovat jim hodnotu přímo v Jinja2 šabloně nebo lze referencovat proměnné, které jsou přiřazeny při renderování šablony. Příklad Jinja2 konstrukce pro přístup k hodnotě referencované proměnné: **{{ variable_name.key }}** (viz obrázek č. 9).
- **Podmínky** - Jinja2 umožňuje definovat **if/elif/else** podmínky (jako v Pythonu). Celá Jinja2 konstrukce podmínky je však ukončena Jinja2 výrazem **{% endif %}** (viz obrázek č. 9).
- **Cykly** - Jinja2 podporuje pouze **for** cyklus. Celá Jinja2 konstrukce for cyklu je ukončena Jinja2 výrazem **{% endfor %}** (viz obrázek č. 9).
- **Filtry** - Pomocí Jinja2 filtrů lze definovat funkce, které slouží k manipulaci dat (přetypování, převod na velká nebo malá písmena atd.). Příklad syntaxe Jinja2 filtru pro převod stringu na malá písmena (lower): **{{ string_variable | lower }}**.

```
{% if static_routing_config is defined %}
{% for net in static_routing_config.networks %}
ip route {{ net.dest_ip }} {{ net.dest_mask }} {{ net.next_hop }}
{% endfor %}
{% if static_routing_config.default is defined %}
ip route 0.0.0.0 0.0.0.0 {{ static_routing_config.default.next_hop }}
{% endif %}
!
end
{% endif %}
```

Obrázek 9: Ukázka šablony konfigurace statického směrování v IPv4 pro směrovače firmy Cisco

1.8 Automatizační nástroje

Automatizační nástroje poskytují ekosystém, který lze využít pro správu serverů a síťových zařízení. Na trhu je k dispozici velké množství automatizačních nástrojů. Nástroje se zejména liší používaným DSL (Domain Specific Language), architekturou nebo způsobem správy jednotlivých zařízení.

Společné rysy automatizačních nástrojů:

- **Ekosystém** - Soubor knihoven/modulů/pluginů, které uživatel může využít k správě konfigurace síťových zařízení.

- **Multithreading** - Nástroje implementují multithreading, tak aby uživatelem definované konstrukce byly prováděny na několika zařízeních současně.
- **Správa inventářů** - Všechny nástroje podporují možnost vytváření inventářů (statických i dynamických). Inventáře jsou soubory, ve kterém jsou specifikována jednotlivá zařízení, skupiny a jejich data.

Hlavní rozdíly mezi nástroji [3]:

- **Agentless vs Agent-based architektura** - U nástrojů řídící se **agentless architekturou** není nutné instalovat a konfigurovat softwarového agenta u spravovaných zařízeních. Příkladem je nástroj Ansible. V případě **agent-based architektury** je nutné nainstalovat a nakonfigurovat softwarového agenta u jednotlivých klientů. Agent-based architekturu využívají nástroje jako například Chef, Puppet a Salt. V rámci automatizace konfigurace síťových zařízení je preferována agentless architektura, především kvůli jednodušší počáteční konfiguraci a absence agenta na straně klienta.
- **DSL vs univerzální programovací jazyk** - **DSL** je jazyk, který se používá pouze v dané doméně. Každý nástroj používá vlastní DSL. Mezi automatizační nástroje používající DSL patří například Ansible, Chef, Puppet a Salt. Existují však i nástroje, které abstrakci pomocí DSL neposkytují, jelikož se s nimi pracuje pomocí **univerzálních programovacích jazyků** (např. Python). Příkladem takového automatizačního nástroje je Nornir.
- **Push vs Pull vs Event-driven model** - U **push modelu** je inicializátorem komunikace centrální server, který zasílá konfigurační data bez nutnosti toho, aby informoval spravované zařízení. Push model využívá nástroj Ansible. **Pull model** se řídí architekturou klient-server. Klient má nakonfigurovaného softwarového agenta. V pravidelných intervalech softwarový agent inicializuje interakci s centrálním serverem, od kterého obdrží konfigurační data a porovná stav nové konfigurace s běžící konfigurací síťového zařízení. Na síťové zařízení jsou potom aplikovány pouze rozdíly v konfiguracích. Pull model využívají automatizační nástroje Chef, Puppet a Salt. **Event-driven automatizační nástroje** využívají **eventy (události)**. Pokud nastane významná událost, tak se provede uživatelem definovaná akce. Příkladem je například nástroj StackStorm.

1.8.1 Ansible

Ansible je open-source automatizační nástroj, který se hlavně používá pro hromadnou správu a orchestraci serverů. Ansible využívá agentless architekturu. Mimo jiné Ansible podporuje push i pull model. Nástroj a jeho moduly jsou napsány v programovacím jazyce Python. První verze nástroje vyšla v roce 2012. Od té doby došlo k mnoha změnám. Součástí verze 2.5 (rok 2018) bylo vydáno hned několik network modulů, které zcela ovlivnily možnosti automatické správy konfigurace směrovačů a přepínačů. Tyto moduly například podporovaly správu síťových zařízení, které mají implementované API. S vývojem multi-vendor Python knihoven se začaly vyvíjet

i multi-vendor moduly (např. napalm-ansible modul). Právě kvůli silné komunitě, propracovaným modulům a celkové jednoduchosti používání nástroje se Ansible stal velmi populární volbou pro hromadnou správu jak serverů tak i ostatních síťových zařízení (směrovačů, přepínačů). [1,3]

Základní koncepty nástroje: [32, 39]

- **Inventory (inventář) - Inventář obsahuje seznam všech definovaných zařízení a skupin**. Inventář lze reprezentovat v datovém formátu INI nebo YAML. V inventáři lze specifikovat i proměnné, které jsou přiřazeny konkrétnímu zařízení nebo skupině (případně globálně). V případě komplexních projektů lze inventář rozdělit, tak že data jednotlivých hostů budou uložena v samostatných souborech. Tyto soubory musí být pojmenovány dle předem definovaného názvu zařízení a uloženy ve složce **host_vars**. Podobný koncept platí i pro data jednotlivých skupin. Jednotlivé soubory musí být uloženy ve složce **group_vars**.
- **Playbooks (scénáře)** - YAML soubory, ve kterém jsou definovány jednotlivé instrukce, které se mají provést na definovaných zařízeních. **Playbook** je složen z jedné nebo více Ansible her (**plays**), přičemž každá Ansible hra je složena z jednoho nebo více Ansible úkolů (**tasks**). V jednotlivých Ansible úkolech jsou potom využity dostupné moduly. Mimo jiné lze jednotlivé Ansible úkoly logicky seskupovat do bloků pomocí atributu **block**.
- **Modules (moduly)** - Moduly jsou Python skripty, které jsou interpretovány na řízeném uzlu (v případě správy serveru) nebo na řídícím uzlu (v případě správy směrovačů a přepinačů). Moduly lze používat v Ansible úkolech.
- **Roles (role)** - Mechanismus, který umožňuje vytvářet nezávislé komponenty. Používá se pro modularizaci Ansible projektu. Role má jasně definovanou adresářovou strukturu. Součástí této struktury jsou složky pro ukládání Jinja2 šablon (templates), Ansible úloh (tasks), konfiguračních souborů (files) atd.´
- **Collections (kolekce)** - Ansible kolekce slouží k distribuci rolí, modulů a scenářů, které jsou vytvořeny třetí stranou.
- **Tags (značky)** - Značky umožňující vybrat pouze konkrétní úkoly nebo hry, které se mají vykonat při spuštění Ansible scénáře.

Typy zařízení v Ansible architektuře: [1, 39]

- **Control node (řídící uzel)** - Linuxový server, na kterém je nainstalovaný Ansible. Pomocí nástroje Ansible spravuje definované řízené uzly. Každý řídící uzel musí být linuxovým serverem s nainstalovaným Python interpretem. Pro interakci s řízenými uzly se používá komunikační protokol SSH. V případě network modulů (modulů pro správu síťových zařízení) lze u podporovaných uzlů komunikovat s implementovaným REST API řízeného uzlu pomocí HTTP/HTTPS protokolu.

- **Managed node (řízený uzel)** - Jakékoliv zařízení, které chceme spravovat nástrojem Ansible (tzn. Windows server, Linux server, směrovače, přepínače). Řízené uzly vyjma síťových zařízení jako např. směrovačů a přepínačů musí mít nainstalovaný Python interpreter. Spravované servery využívají Python interpreter pro spuštění jednotlivých modulů. V případě správy síťových zařízení (směrovačů, přepínačů) jsou network moduly interpretovány lokálně (v řídícím uzlu), jelikož spravované zařízení nemusí mít nutně k dispozici Python interpreter.

Základní struktura Ansible hry (viz výpis č. 1):

- name - Určuje název Ansible hry. Pokud je atribut **name** uveden pod atributem **task**, tak určuje název Ansible úkolu.
- hosts - Pomocí atributu **hosts** lze definovat na jakých uzlech (zařízeních) budou provedeny dané úkoly. Uživatel zde může definovat celé skupiny (servers) i jednotlivá zařízení (host1). Dále lze využít i regulárních výrazů. Definované skupiny a zařízení musí být však v souladu s vytvořeným inventářem.
- gather_facts - Používá se pro zjištění základních údajů o cilových zařízeních (např. operační systém, IP adresy síťových rozhraní atd.).
- vars - Pomocí atributu **vars** lze definovat proměnné, které lze referencovat pouze v dané Ansible hře.
- tasks - Hodnotou **tasks** atributu je list jednotlivých Ansible úkolů.
- debug - Modul, který se používá pro výpis obsahu proměnných do konzole. Obsahuje atribut **msg**, pomocí kterého lze specifikovat string, který bude vypsán do konzole.
- "{{ description }}"- Jínja2 výraz pro přístup k hodnotě proměnné description.

```
- name: Print variable
  hosts: servers,host1
  gather_facts: no
  vars:
   description: value
  tasks:
   - name: Print variable description
     debug:
        msg: "{{ description }}"
```

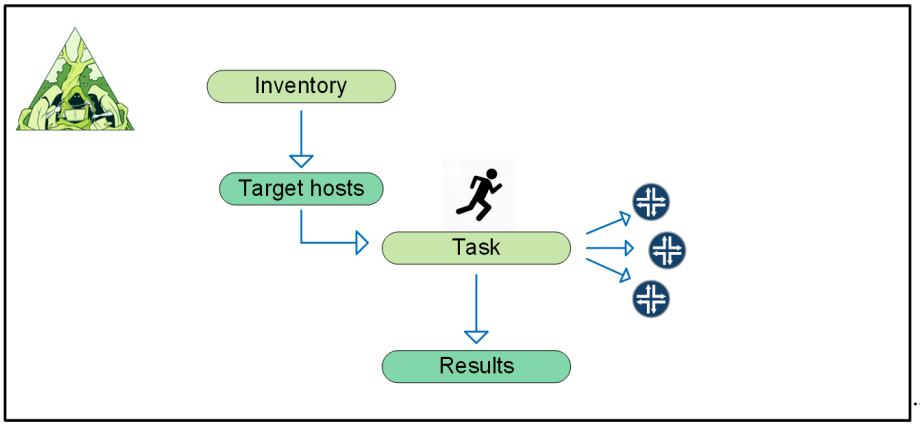
Výpis 1: Struktura Ansible hry

1.8.2 Nornir

Nornir je framework, který byl speciálně vyvinut pro správu síťových zařízení v programovacím jazyce Python. Ve srovnání s ostatními automatizačními nástroji Nornir nevyužívá DSL pro práci s nástrojem [42] ale programovací jazyk Python. Uživatel má tak možnost využívat celý ekosystém programovacího jazyka Python včetně IDE a nástrojů na ladění kódu. První oficiální vydání frameworku byl v roce 2017 (v0.0.1). V prvotních verzích (v0.0.x, v1.x a v2.x) byl framework velmi robustní. V rámci instalace frameworku byly instalovány všechny knihovny, které jsou potřebné k automatizování síťových konfigurací v programovacím jazyce Python (např. knihovny jinja2, paramiko, netmiko, napalm, netbox, py-junos-eznc atd.). [43, 44]

Nejnovější verze (v3.0.0) vyšla v roce 2020 a zcela pozměnila architekturu frameworku. Od nové verze knihovna nornir implementuje pouze nejdůležitější funkcionalitu automatizačního nástroje (multithreading, správu inventáře), přičemž pro konfiguraci a interakci se síťovými zařízeními se využívají už implementované knihovny. Nová verze frameworku však přistupuje k jednotlivým knihovnám jako k pluginům, které je možné instalovat dle potřeby uživatele pomocí správce Python balíčků **PIP** (PIP Installs Packages). [44, 45]

Princip frameworku Nornir (viz obrázek č. 10) spočívá v definování inventáře (Inventory) a vytvářením úkolů (Tasks). V inventáři specifikujeme jednotlivé hosty (síťová zařízení) a jejich data, která budou použita v úkolech. Úkoly jsou funkce, jejichž prvním argumentem je tzv. Task objekt. Task objekt obsahuje zparsované data z inventáře a má možnost volat další podúkoly. Úkoly jsou implementovány uživatelem tak, jako kdyby je chtěl volat pouze na jedno konkrétní síťové zařízení. Nornir však tyto úkoly potom provede paralelně (dle inventáře a použitého filtrování). Po provedení celého úkolu jsou výsledky ze všech síťových zařízení agregovány podle názvu síťového zařízení. [46]



Obrázek 10: Princip frameworku Nornir [46]

Nornir umí pracovat s několika typy inventářů: [44, 46]

- **SimpleInventory** - Defaultní Nornir inventář, který se skládá ze tří souborů (**hosts.yaml**, **groups.yaml** a **defaults.yaml**). V **hosts.yaml** jsou definovány data a parametry jednotlivých síťových zařízení. **Groups.yaml** slouží k definování skupin a dat které jim náleží. V **defaults.yaml** lze definovat globální parametry, které náleží všem zařízením.
- **AnsibleInventory** - Plugin používající se pro parsování základního Ansible inventáře, který je vytvořen v YAML formátu. Nepodporuje INI formát, group_vars a host_vars.
- **NetBoxInventory2** - Plugin, který se používá pro interakci s webovou aplikací NetBox za účelem vytvoření dynamického inventáře.
- **IPFabricInventory** - Plugin, který lze využít pro interakci s implementovaným REST API nástroje IP Fabric za účelem vytvoření dynamického inventáře. [47]

Přehled nejpoužívanějších pluginů: [44]

- **nornir_napalm** - Plugin obsahující Python knihovnu **napalm**, která je optimalizována pro potřeby Nornir frameworku.
- **nornir_netmiko** - Součástí pluginu je rozšíření Python knihovny **netmiko**, které má implementované Nornir úkoly.
- **nornir_utils** - Série pluginů, které implementují pomocné a často využívané metody. Implementované jsou například tyto metody: paralelní načtení dodatečných YAML a JSON souborů (load_yaml, load_json) případně výpis výsledků z Nornir úkolů do konzole (print_result).
- **nornir_jinja2** - Plugin obsahující Python knihovnu **jinja2**, která je optimalizována pro potřeby Nornir frameworku. Umožňuje paralelně renderovat Jinja2 šablony.

Na rozdíl od ostatních nástrojů je Nornir výrazně rychlejší při správě většího množství zařízení (>=100 zařízení) [41]. Pro nástroj nejsou momentálně vyvinuty pluginy, které by umožnily deklarativní přístup ke konfiguraci serverů.

1.8.3 StackStorm

StackStorm je open source nástroj, který se používá pro event-driven automatizaci. Ve srovnání s ostatními automatizačními nástroji StackStorm monitoruje jednotlivá zařízení a reaguje na patřičné eventy (události). StackStorm lze využít pro správu síťových zařízení a serverů včetně jejich monitorování. Mimo jiné StackStorm poskytuje interakci s dalšími externími službami a nástroji jako například Nagios a Ansible. Kombinací těchto nástrojů lze například provést hromadnou konfiguraci síťových zařízení za předpokladu, že se stane nějaká významná událost. [3]

Základní koncepty nástroje: [48, 49]

- **Sensors** - Senzory jsou Python skripty, které periodicky sbírají data ze síťových zařízení a serverů. V případě rozsáhlé infrastruktury jsou data sbírána z externího systému. Senzory umožňují definovat triggery, které jsou aktivovány v případě, že nastane nějaká významná událost (event).
- **Triggers** - Triggery reprezentují StackStorm události. Triggery jsou aktivovány v případě, že nastane významná událost na základě obdržených dat z externího systému. Uživatel může taky vytvářet vlastní triggery.
- **Actions (akce)** - StackStorm akce, které se mají provést v případě, že nastane event. Akcí může být například spuštění určitého Python skriptu nebo Ansible scénáře.
- **Rules** - Pravidla, která mapují triggery (triggers) k jednotlivým StackStorm akcím (actions). Pomocí pravidel lze tedy definovat, které akce se provedou při aktivaci daného triggeru.
- **Workflows** - Seskupení souvisejících StackStorm akcí.
- **Packs** - Používají se pro distribuci StackStorm akcí, workflows, pravidel, senzorů a triggerů.

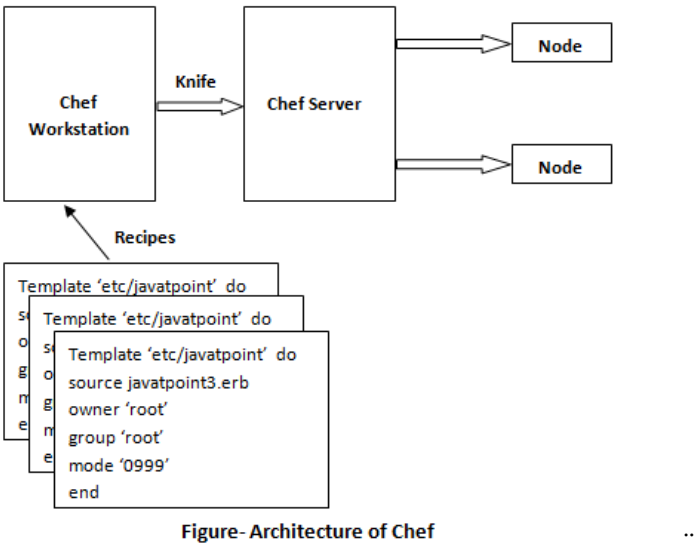
1.8.4 Chef

Open source nástroj, který se používá hlavně pro správu serverů. Chef je napsaný v jazyce Ruby a používá specifické DSL pro vytváření receptů. Na rozdíl od nástroje Ansible se Chef řídí architekturou klient-server a využívá tzv. pull model pro dynamickou aktualizaci stavu klientských nodů. Jelikož Chef využívá pull model, tak je nutné na všech spravovaných zařízeních (tzv. Chef nodes) dodatečně nainstalovat a nakonfigurovat softwarového agenta Chef Client. Z toho důvodu není Chef moc používaný pro správu síťových zařízení (směrovačů, přepínačů). [32, 50]

Základní koncepty nástroje: [50, 51]

- **Recipes (recepty)** - Úkoly, které se mají na klientských uzlech provést (např. instalace softwarových balíčků). Recepty jsou psány v programovacím jazyce Ruby. Recept může být závislý na jiných receptech. Pořadí ve kterém budou recepty vykonány potom určuje run-list. Každý recept, který chceme aby byl vykonán klientským uzlem, tak musí být v run-listu definován.
- **Cookbooks (kuchařky)** - Znovupoužitelná komponenta, jenž se skládá z receptů, pravidel, knihoven, metadat a dalších prostředků, které jsou nutné pro správu klientských uzlů.

- **Run-lists** - Run-list slouží k specifikování pořadí, v jakém budou vykonány jednotlivé recepty. Mimo jiné run-list obsahuje i informaci o tom, na kterých klientech má být daný recept proveden. Všechny run-listy jsou definované na centrálním serveru.



Obrázek 11: Architektura nástroje Chef [50]

Typy uzlů: [50, 51]

- **Chef Server** - Centrální server, ve kterém jsou uloženy kuchařky a konfigurační data.
- **Chef Workstation** - Počítač, na kterém vytváříme všechny kuchařky a jejich recepty. Pomocí nástroje Knife lze kuchařky přesunout na Chef Server (viz obrázek č. 11).
- **Chef Nodes** - Klientské uzly, které mají nakonfigurovaného softwarového agent Chef Client. V uživatelem definovaných časových intervalech Chef Client obdrží z Chef serveru potřebné konfigurační data a kuchařky. Kuchařky se potom aplikují na samotných klientských uzlech (viz obrázek č. 11).

1.9 Verzovací systémy

Systém správy verzí je systém, který umožňuje evidovat změny v sledovaných souborech. V případě automatizování síťových konfigurací se může jednat o definované inventáře, Jinja2 šablony, zdrojové kódy, konfigurace nebo Ansible scénáře. Verzovací systémy umožňují zaznamenat jakoukoliv modifikaci v sledovaném souboru.

Systémy správy verzí se hlavně používají pro práci v týmech. Uživatelé mohou sledované změny komentovat a vytvářet revize pro ostatní členy týmu. Ostatní členové týmu si mohou takto aktualizovat svou pracovní kopii projektu. Při aktualizaci své pracovní kopie může docházet

ke kolizím, které je nutné potom manuálně vyřešit. Dále systém správy verzí umožňuje uživateli přecházet mezi jednotlivými revizemi. Na jednotlivé revize je použita hashovací funkce pro úsporu místa. Pro každou revizi nejsou vytvořené kopie jednotlivých souborů, ale zaznamenávají se pouze změny mezi revizemi. Názvy a typy operací se liší dle použitého verzovacího systému.

1.9.1 Centralizované verzovací systémy

Centralizované verzovací systémy využívají architekturu typu client-server. Uživatelé využívají pouze jeden společný repozitář (tzv. centrální repozitář). Do centrálního repozitáře se potom ukládají změny od jednotlivých uživatelů. Nejpoužívanějším centralizovaným verzovacím systémem je systém SVN (Subversion). Největší výhodou je jednoduchost při práci s takovým systémem. Hlavní nevýhodou je potom samotná architektura systému. Pokud nastane problém se serverem, kde je centrální repozitář nakonfigurován, tak uživatelé nemají přístup k celé historii (všem revizím) daného projektu.

1.9.2 Decentralizované verzovací systémy

Decentralizované verzovací systémy adresují problémy centralizovaných verzovacích systémů. Každý uživatel má svůj vlastní lokální repozitář, který obsahuje celou historii daného projektu (v případě, že je lokální repozitář pravidelně synchronizován se sdíleným repozitářem). Všechny změny, které provede uživatel, jsou nejdříve evidovány v lokálním repozitáři. Sdílený repozitář (např. vytvořený pomocí platformy GitHub) slouží pro evidování změn z jednotlivých lokálních repozitářů. Změny ostatních uživatelů si lze aktualizovat synchronizací sdíleného a lokálního repozitáře. Z lokálního repozitáře lze patřičné změny sloučit s aktuální pracovní kopií jiného uživatele. Další významnou funkcí je možnost větvení. Větve umožňují uživatelům efektivně pracovat na různých částech projektu bez zásahu do hlavní větve. Jednotlivé větve lze potom slučovat s hlavní větví. Momentálně nejpoužívanějším decentralizovaným verzovacím systémem je Git. [52, 53] Největší výhodou decentralizovaných verzovacích systémů je především možnost pracovat s vlastním lokálním repozitářem (není nutné se neustále spoléhat na konektivitu k sdílenému repozitáři). Další výhodou je možnost větvení. Nevýhodou je však celková náročnost při práci s decentralizovanými verzovacími systémy (příkazy, pracovní postupy).

2 Monitorování síťových zařízení

V dnešní době je zákazníky vyžádováno, aby poskytované služby byly vždy dostupné. Pro garanci co nejvyšší dostupnosti služeb je nutné mít přehled o síťovém provozu, stavu jednotlivých zařízení a poskytovaných službách. Monitorováním síťových zařízení lze získat přehled o aktuálním stavu daných zařízení. Pokud by nastal nějaký problém v síťové infrastruktuře, tak efektivním monitorováním lze vzniklé problémy rychle diagnostikovat a odstranit. Monitorovat lze například vytížení procesoru a paměti zařízení, stav síťových rozhraní, dostupnost serverů a služeb atd. V posledních letech se začal využívat nový přístup k monitorování síťových zařízení tzv. **Streaming telemetry**, který adresuje problémy protokolu **SNMP** [54].

2.1 Simple Network Management Protocol

SNMP je standardizovaný protokol, který byl představen v roce 1988. SNMP lze použít pro konfiguraci a **sběr dat** ze síťových zařízení. Poslední verze protokolu je SNMPv3, která adresuje problémy zabezpečení a autentifikace. SNMP defaultně používá pro přenos datagramů protokol UDP (User Datagram Protocol). SNMP architektura je složena ze **SNMP manažera** a **SNMP agenta**. **SNMP manažer** je program, který je instalován na monitorovacím serveru. SNMP manažer zasílá dotazy SNMP agentovi (pull přístup) a zároveň prezentuje obdržené data. **SNMP agent** je softwarový klient nakonfigurovaný na síťovém zařízení, který sbírá data o daném síťovém zařízení a jeho stavu. SNMP agent reaguje na dotazy SNMP manažera (např. SNMP dotazy GET, SET) a dále umožňuje odesílat SNMP manažerovi TRAP zprávy bez vyžádání manažera. Defaultně je používán UDP port 161 pro příjem dotazů od SNMP manažera a UDP port 162 pro příjem TRAP zpráv. [1, 55] Shromážděné data síťového zařízení jsou hierarchicky strukturována v **MIB** (Management Information Base) databázi. MIB databáze má stromovou strukturu, která je tvořena jednoznačně identifikovatelnými objekty. Každý takový objekt lze identifikovat pomocí identifikátoru objektu **OID** (Object identifier). [56]

Nevýhody SNMP protokolu: [54, 57]

1. **SNMP manažer se dotazuje SNMP agentů o data v daných časových intervalech. Pomocí SNMP protokolu lze tedy získat určitý přehled o stavu síťových zařízení. Na druhou stranu pomocí SNMP protokolu nelze spolehlivě monitorovat stav zařízení v reálném času.**
2. Nedostatečné zabezpečení přenosu SNMP zpráv (týká se verzí SNMPv1 a SNMPv2).
3. SNMP protokol není vhodný pro komplexní síťové infrastruktury (malá škálovatelnost). S narůstajícím počtem síťových zařízení narůstá počet SNMP dotazů, které jsou zaslány SNMP agentům a množství dat, které musí SNMP manažer zpracovat. **U SNMP manažera se tak může razantně navýšit vytížení procesoru a RAM (Random-access**

memory) paměti. Zpracováváním SNMP dotazů je vytížen procesor i operační pamět jednotlivých zařízení.

4. Některé informace o stavu zařízení nemusí být dostupné v MIB databázi. Takové informace mohou být dostupné pouze použitím CLI.

2.2 Streaming telemetry

Streaming telemetry představuje nový přístup k monitorování síťových zařízení. Ústředním konceptem je **telemetrie**, kterou lze definovat jako automatizovaný komunikační proces, v rámci kterého jsou data monitorovaných síťových zařízení vysílána („streamována") softwarovému agentovi (např. Telegrafu), který posbírá data a vloží je do datového úložiště (např. InfluxDB). Oproti SNMP protokolu využívá telemetrie tzv. subscription model. **Subscription** je nastaven na monitorovaném síťovém zařízení. Subscription slouží k automatickému vysílání („streamování") strukturovaných dat síťového zařízení. [58] Uživatel může pro každý subscription nastavit, která data se mají vysílat. Dále lze nastavit časový interval pro vysílání dat, zdrojovou IP (Internet Protocol) adresu, cílovou IP adresu příjemce [59]. Na rozdíl od SNMP využívá Streaming telemetry tzv. „push"model. **U Streaming telemetry neexistuje žádný centrální prvek, který se v určitém časovém intervalu dotazuje jednotlivých agentů na data. Namísto toho každé síťového zařízení vysílá určitá data dle potřeby uživatele.** Streaming telemetry lze využít pro monitorování síťových zařízení téměř v reálném čase. K dispozici jsou různé implementace telemetrií. Při implementaci lze využít různé transportní protokoly jako např. gRPC, TCP (Transmission Control Protocol) nebo UDP. [54]

V tuto chvíli není telemetrie standardizována. Na základě zvyšující se popularity YANG modelů se však změnila i implementace telemetrie. Telemetrie používající YANG modely patří do skupiny **Model-Driven Telemetry**. Vytvořený subscription vysílá data, jejíchž struktura je definována YANG modelem. [58]

2.3 Časové řady

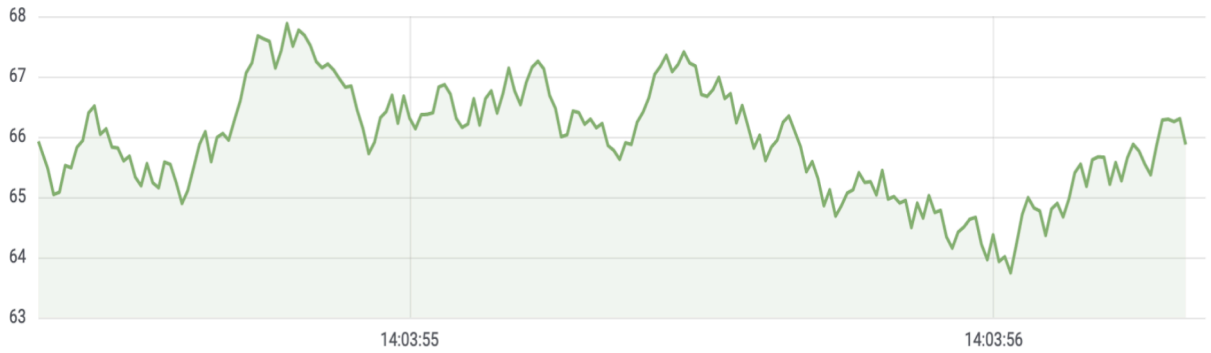
Časová řada je tvořena sekvencí časových bodů, které jsou sbírány v určitých časových intervalech. Data jsou chronologicky řazena podle času, kdy bylo měření provedeno. Pomocí dat časové řady lze sledovat změny v průběhu času. V rámci jednoho měření je nutné vždy zaznamenat informaci o měřené hodnotě a o časové značce, která vyjadřuje, kdy bylo dané měření provedeno. [60]

Příklady použití časových řad: [61]

- **IoT (Internet of Things)** - Můžeme zde zařadit data, která jsou získána ze senzorů. Příkladem je měření teploty, vlhkosti vzduchu, spotřeby energie, atmosférického tlaku atd.

- **Ekonomika** - Vývoj hodnoty akcií společnosti, vývoj ceny produktů atd.
- **Stav síťových služeb a zařízení** - Vytížení procesoru a RAM paměti, logování událostí.

Data časové řady jsou v pravidelných intervalech ukládána do databáze časových řad. Data časové řady mohou reprezentovat i nepravidelné údálosti. Data časové řady lze nejlépe vizualizovat formou grafu. Graf (viz obrázek č. 12) zobrazuje měřenou hodnotu (svislá osa) v závislosti na čase provedeného měření (vodorovná osa). V případě analýzy trendů v datech lze hodnoty časové řady agregovat za určité časové období [61].



Obrázek 12: Ukázka grafu časové řady [61]

2.4 Databáze časových řad

TSDB (Time Series Database) jsou databáze, které jsou určeny pro ukládání dat časových řad. Na rozdíl od relačních databází nejsou databáze časových řad univerzální. Velká část TSDB databází využívá flexibilitu NoSQL databází, která je rozšířena optimalizovaným ukládáním dat časových řad a jejich vybíráním. Ve srovnání s NoSQL databází lze však ukládat do TSDB databází pouze data časových řad. [62]

2.4.1 InfluxDB

InfluxDB je open source databáze časových řad, která je napsána v programovacím jazyce Go. Součástí InfluxDB je implementované REST API, které je používáno při zapisování dat do databáze a jejich vyhledávání. [63] Pro psaní dotazů podporuje InfluxDB dva dotazovací jazyky: InfluxQL a Flux. Syntaxe dotazovacího jazyka InfluxQL je velmi podobná dotazovacímu jazyku SQL (Structured Query Language). Na druhou stranu dotazovací jazyk Flux je zcela odlišný od jazyka SQL. Pro zápis a uložení datových bodů využívá InfluxDB tzv. **line protocol**. Line protokol je textový serializační formát, který odděluje jednotlivé elementy protokolu mezerou (vyjma měření), přičemž každý nový datový bod je na novém řádku. [64, 65]

Elementy line protokolu: [65]

- **Measurement (povinné)** - Measurement reprezentuje dané měření (např. vytížení procesoru). Název měření je reprezentován jako string. V měření jsou uložena naměřená data.
- **Tag set (volitelné)** - Kolekce tagů (značek), přičemž každý tag je reprezentován párem klíč=hodnota. Klíč i hodnota musí být reprezentovány jako textový řetězec. Data měření jsou na základě tagů indexována. Tagy jsou metadata, která lze použít pro efektivní filtrování a výběr dat.
- **Field set (povinné)** - Kolekce datových polí (fields) daného datového bodu, přičemž každé datové pole je definováno párem klíč=hodnota. Klíč musí být textový řetězec. U hodnoty jsou podporovány následující datové typy: Float, Integer, UInteger, String a Boolean. Datová pole nejsou indexována. Každý datový bod musí mít definován minimálně jedno datové pole.
- **Timestamp (volitelné)** - Unixový čas, který reprezentuje v jakém časovém okamžiku bylo měření provedeno. Pokud není timestamp specifikován, tak se použije systémový čas serveru (v nanosekundách).

2.4.2 OpenTSDB

OpenTSDB je databáze časových řad, která je nástavbou NoSQL (Not Only SQL) databáze HBase. HBase je distribuovaná a sloupcová NoSQL databáze, která ukládá data do databáze ve formě sloupců místo řádků [66]. OpenTSDB databáze je nástavbou HBase databáze optimalizovanou pro ukládání a správu dat časových řad. Architektura OpenTSDB databáze je složena z TSD (Time Series Daemon) serverů a HBase databáze. TSD umožňuje vkládat data do HBase databáze nebo vybírat data z databáze. TSD implementuje REST API, které klient může použít pro interakci se službou. Největší výhodou OpenTSDB je škálovatelnost. HBase databáze je horizontálně škálovatelná a TSD servery jsou na sobě zcela nezávislé. [67]

2.4.3 TimescaleDB

TimescaleDB je rozšíření PostgreSQL systému pro správu relačních databází, které je optimalizované pro ukládání a správu dat časových řad. TimescaleDB nativně podporuje dotazovací jazyk SQL. Uživatelé, kteří někdy používali některý relační databázový systém (např. MySQL, PostgreSQL), tak nebudou mít problém používat i TimescaleDB. Vytváření relace funguje úplně na stejném principu, jako u PostgreSQL databáze. TimescaleDB relace však musí obsahovat jeden atribut, který musí nabývat datového typu **timestamp** nebo **date**. Ve srovnání s relační databází využívá TimescaleDB konceptu hypertabulky, kterou lze z vytvořené relace vytvořit pomocí funkce **create_hypertable**. [68]

Hypertabulka je abstrakní struktura, která je složena z oddílů (tzv. **chunks**). Každý oddíl (**chunk**) je tvořen relací, která obsahuje uložená data pouze v nějakém časovém intervalu.

Rozdělení hypertabulky na dané oddíly je prováděno automaticky samotným databázovým systémem. Rozdělení hypertabulky je vždy prováděno podle atributu s datovým typem **timestamp** nebo **date**. Uživatelé při vytváření SQL dotazů používají samotnou hypertabulku. [69]

TimescaleDB adresuje problémy relačních databází při ukládání a správě dat časových řad. Zároveň však nabízí uživatelům prostředí a možnosti, na které jsou zvyklí při používání relačních databází. Mimo jiné TimescaleDB nabízí možnost databázi horizontálně škálovat.

2.5 Telegraf

Telegraf je open source agent napsaný v programovacím jazyce Go. [72] Agenta lze nakonfigurovat například na serverech nebo Cisco zařízeních s operačním systémem IOS XE. Telegraf se používá ke sběru, zpracování a agregování dat časových řad. Mimo jiné agregované data lze v pravidelných intervalech vkládat do databáze. [73] Tuto funkcionalitu zajišťuje řada pluginů, které lze specifikovat v konfiguračním souboru agenta.

Typy pluginů: [74]

- **Input** - Input pluginy slouží ke sběru dat časových řad. Pomocí input pluginů lze sbírat například data o komponentách samotných zařízení (vytížení procesoru atd.), o stavu síťových rozhraní nebo služeb serveru (Nginx, MySQL atd.). [73]
- **Output** - Output pluginy umožňují vkládat data časové řady do TSDB databáze (například InfluxDB nebo OpenTSDB). [73]
- **Aggregator** - Aggregator pluginy agregují data časové řady. Příkladem je například **BasicStats** plugin, který agreguje data pomocí agregačních funkcí (aritmetický průměr, maximum atd.).
- **Processor** - Processor pluginy umožňují zpracovávat, filtrovat a transformovat data.
- **External** - Externí pluginy jsou pluginy, které nejsou součástí Telegraf ekosystému. Pluginy nemusí být implementovány v programovacím jazyce Go.

2.6 Grafana

Grafana je open source platforma, která se zejména používá pro vizualizaci dat časových řad. Grafana umožňuje vhodně prezentovat data formou grafů. Pomocí nástroje lze vytvářet komplexní databázové dotazy, pomocí kterých lze vybrat data z databáze a vizualizovat je. Defaultně podporuje Grafana řadu databázových systémů, mezi které například patří MySQL, InfluxDB, Prometheus atd. Seznam podporovaných databázových systémů lze dále rozšířit pomocí pluginů. Nástroj podporuje i zasílání výstražných notifikací na základě předem definovaných pravidel. [70]

Základní koncepty platformy: [71]

- **Dashboard** - Dashbord je tvořen panely, které jsou rozmístěny do mřížky. V dashboardu lze zejména přidávat nové panely a nastavovat parametry, které budou platné pro všechny panely (např. definování časového intervala, doba refreshování panelů atd.).
- **Panel** - Panel tvoří základní blok pro vizualizaci dat. Panely jsou umístěny v dashboardu. Každý panel nabízí query editor pro vytváření databázových dotazů. Uživatel může nastavit velikost panelu, dobu refreshování panelu, časový interval atd.
- **Data sources** - Datové zdroje, se kterými Grafana komunikuje. Data z datového zdroje jsou vizualizovány pomocí Grafany. Grafana například podporuje MySQL, InfluxDB, Prometheus atd.
- **Query editor** - Formulář, který zjednodušuje psaní základních databázových dotazů. Při tvorbě komplexních dotazů lze dotazy vytvářet zcela manuálně.
- **Plugin** - Pluginy, které rozšiřují funkcionalitu platformy. Příkladem je rozšíření datových zdrojů, nastavení panelů atd.

2.7 Monitorovací nástroje

Monitorovací nástroje představují kompletní řešení pro monitorování síťových zařízení, serverů a jejich provozovaných služeb. Nástroje umožňují sbírat data z jednotlivých zařízení, ukládat je do databáze a vizualizovat je pro uživatele. Nástroje dále podporují výstražné notifikace, které lze například použít v situacích, kdy data překročí hraniční hodnotu. Jednotlivé nástroje se liší především použitou architekturou, možnostmi nastavení a pluginy.

2.7.1 Nagios

Monitorovací nástroj, který se zejména používá pro monitorování síťových zařízení, serverů a provozovaných síťových služeb. Nagios je dostupný ve dvou variantách. První variantou je Nagios Core, který je open source. Nagios Core poskytuje základní funkcionalitu pro monitorování zařízení a služeb (monitoring, notifikace, pluginy). Druhá varianta Nagios XI je placená a tvoří nástavbu Nagios Core. Nagios XI především poskytuje komplexní webové rozhraní, které lze využít pro vizualizaci dat nebo nastavení monitorovaných služeb. [75] Monitorování zařízení a služeb je prováděno pomocí pluginů. Pluginy jsou externí komponenty, které umožní posbírat potřebná data, analyzovat je a vrátit výsledek formou statusu a detailního popisu statusu služby [76].

Architektura nástroje Nagios [77]:

- **Nagios server** - Centrální server, který plánuje spouštění pluginů na monitorovaných zařízeních. Mimo jiné obsahuje prosté databázové soubory a webové rozhraní.
- **Databáze** - Defaultně nástroj Nagios ukládá veškerá obdržená data do prostých databázových souborů, které jsou vytvořeny na Nagios serveru. Uživatel však může přidat rozšíření NDOUTILS (Nagios Data Output Utilities), které umožňuje získané data ukládat do MySQL databáze [78].
- **Monitorovaná zařízení** - Zařízení, která chceme monitorovat. Plugin shromáždí potřebná data, ověří je, analyzuje a výsledek zašle zpět Nagios serveru.
- **Grafické uživatelské rozhraní** - V případě Nagios Core je GUI (Graphic User Interface) tvořeno CGI (Common Gateway Interface) skripty. Nagios XI poskytuje webovou aplikaci, kterou lze využít pro vizualizaci dat nebo nastavení monitorovaných služeb.

2.7.2 Zabbix

Zabbix je open source monitorovací nástroj, který se používá pro monitorování síťových zařízení, serverů, aplikací a síťového provozu. Zabbix umožňuje sbírat data ze síťových zařízení pomocí SNMP protokolu. Získané data jsou uložena v relační databázi (MySQL, PostgreSQL, SQLite). Součástí nástroje je i webové rozhraní, které umožňuje graficky vizualizovat uložená data. [79]

Zabbix architektura je složena z následujících komponent: [80]

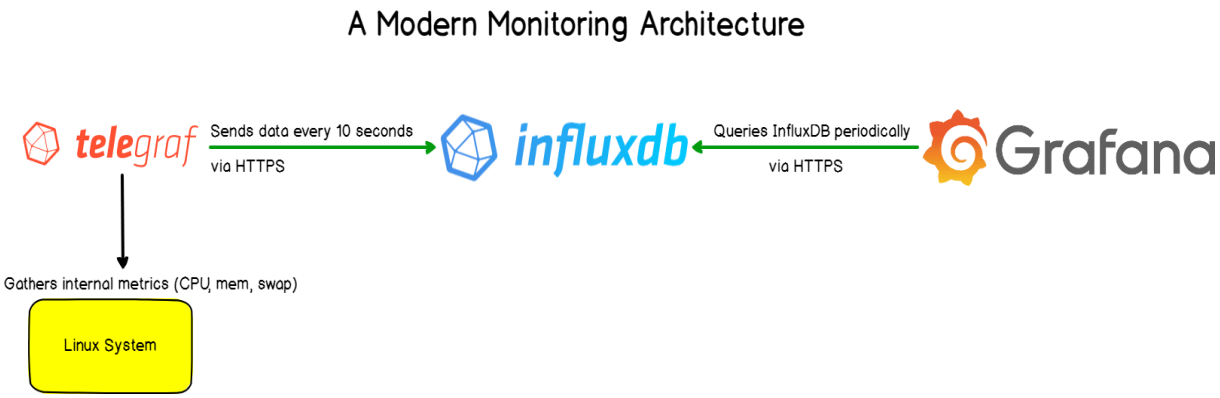
- **Zabbix server** - Centrální komponenta, která přijímá data ze Zabbix agentů. Na Zabbix serveru jsou uložena veškerá obdržená data. Zabbix server nelze zprovoznit na operačním systému Windows.
- **Databáze** - Databáze se nachází na Zabbix serveru a slouží jako úložiště získaných dat. Zabbix podporuje pouze relační databáze (konkrétně MySQL, PostgreSQL, SQLite).
- **Webové rozhraní** - Webová aplikace, která se především používá jako rozhraní pro vizualizaci uložených dat. Webové rozhraní je součástí Zabbix serveru.
- **Zabbix agent** - Softwarový agent, který je konfigurován na zařízeních, které chceme monitorovat. Agent monitoruje zařízení a zasílá Zabbix serveru nebo Zabbix proxy shromážděná data.
- **Zabbix proxy** (volitelné) - Komponenta, která přijímá data ze Zabbix agentů. Pomocí Zabbix proxy lze optimalizovat zátěž Zabbix serveru.

Zabbix Items umožňují definovat testy, které se v definovaných časových intervalech provedou na Zabbix agentech. Agenti potom informují Zabbix server nebo Zabbix proxy o výsledcích daných testů. **Zabbix triggery** monitorují výsledky jednotlivých testů. Pokud test překročí

předem definovanou hraniční hodnotu, tak se aktivuje trigger. Po aktivaci triggeru (při změně stavu dat) se mohou provést uživatelem definované **akce (actions)**. [79]

2.7.3 TIG (Telegraf, InfluxDB, Grafana) stack

TIG stack slouží k monitorování serverů a síťových zařízení téměř v reálném čase. Architektura TIG stacku (viz obrázek č. 13) se skládá ze tří komponent: **Telegraf** (viz oddíl č. 2.5), **InfluxDB** (viz oddíl č. 2.4.1), **Grafana** (viz oddíl č. 2.6). Telegraf sbírá data ze zařízení a vkládá je do InfluxDB databáze. Pomocí Grafany lze vybrat potřebná data z InfluxDB a vhodně je vizualizovat pomocí grafu. [81] Nevýhodou tohoto stacku je především v konfiguraci a zprovoznění agenta, který není u některých zařízení podporován.



Obrázek 13: Architektura TIG stack řešení [82]

3 Analýza automatizovaného systému

3.1 Analýza požadavků na automatizovaný systém pro konfiguraci a monitorování síťových zařízení

Na navržený automatizovaný systém je kladeno několik požadavků. Automatizovaný systém by měl umět konfigurovat a monitorovat síťová zařízení různých vendorů. Automatizovaný systém by měl být zaměřený na síťová zařízení, která používají pouze CLI. Mimo jiné systém by měl umět konfigurovat a monitorovat různé typy síťových zařízení (přepínače, směrovače). Vytvořený systém by měl být zejména flexibilní (konfigurace různých směrovacích protokolů). Dále by měl systém nabízet možnost konfigurace síťových zařízení v IPv4 i IPv6 (Internet Protocol version 6) prostředí. Základní uživatelské požadavky na část automatizovaného systému pro hromadnou konfiguraci jsou mapovány pomocí UML (Unified Modeling Language) diagramu případů užití (viz příloha A).

Jednotlivé případy užití části automatizovaného systému pro hromadnou konfiguraci síťových zařízení:

1. **Konfigurace síťových rozhraní** - Konfigurace fyzických portů a SVI (Switch Virtual Interface), konfigurace základních vlastností portů (popis, rychlost, duplex, IP adresa). Měla by fungovat i konfigurace názvů VLAN (Virtual LAN) a přiřazení switch portu do access nebo trunk módu.

2. **Konfigurace NAT (Network address translation)** - Konfigurace dynamického source NAT.

3. **Konfigurace paketových filtrů** - Konfigurace názvů paketových filtrů a definování jednotlivých pravidel.

4. **Konfigurace Ubuntu serverů** - Konfigurace FTPS (FTP Secure) serveru a TIG stacku.

5. **Mazání části konfigurace** - Možnost smazat konkrétní části síťového konfigurace (např. statické směrování).

6. **Vytváření reportů** - Vytváření základních souhrnných reportů (informace o jednotlivých zařízení, statistiky přijímaných a vysílaných paketů na síťových rozhraní jednotlivých zařízení).

7. **Export dat do .txt a .conf souborů** - Exportování obsahu běžící konfigurace, směrovacích tabulek nebo údajů o vytvořených paketových filtrech.

8. **Zálohování síťové konfigurace** - Možnost provést zálohu síťové konfigurace.

9. **Obnovení síťové konfigurace** - Obnovit konfiguraci z předem vytvořené zálohy dle uživatelem specifikovaného data.

10. **Sběr dat ze síťových zařízení** - Sběr dat ze síťových zařízení a jejich výpis do konzole. Uživatel by měl mít možnost vypsat běžící konfiguraci, obsah směrovací tabulky, informace o jednotlivých síťových rozhraní a další.

11. **Konfigurace směrovacích protokolů** - Uživatel by měl mít možnost konfigurace statického směrování nebo možnost využít směrovací protokoly OSPF (Open Shortest Path First), EIGRP (Enhanced Interior Gateway Routing Protocol) včetně redistribuce statických cest a směrovacích informací z jiných směrovacích protokolů. U EIGRP a OSPF by měla být možnost konfigurace pasivních rozhraní.

Pro monitorovací část systému platí, že by systém měl být schopný monitorovat síťová zařízení a linuxové servery. Monitorovací část systému musí umět pravidelně sbírat potřebná data, uložit je do databáze a z databáze vybrat data, která budou vizualizována.

Co by měl systém sledovat:

1. **Uptime** - dostupnost síťových zařízení a serverů

2. **Vytížení hardwarových prostředků** - vytížení procesoru

3.2 Výběr technologií a nástrojů pro implementaci a testování obou řešení

Výsledný automatizovaný systém bude implementován pomocí dvou odlišných automatizačních nástrojů. První implementace bude provedena pomocí nástroje **Ansible**. Druhé řešení bude provedeno pomocí Python frameworku **Nornir**. Systém bude zejména využíván pro síťová zařízení, která využívají operační systém Cisco IOS nebo Junos OS (Junos operating system). Obě řešení by měly být taky použitelné i pro zařízení s operačním systémem Ubuntu 18.04. Pro konfiguraci a sběr dat budou použity **moduly/tasky** využívající knihovnu **NAPALM**. Sběr dat bude implementován zejména pomocí **NAPALM getterů**. Pokud NAPALM getter nebude implementován pro specifikovaný požadavek, tak bude nutné definovat přímo příkazy, které budou na síťových zařízení vykonány pomocí tasku/modulu **napalm_cli** nebo modulů/tasků založených na knihovně **Netmiko**. Pro vytváření šablon konfiguračních souborů bude použit šablonovací systém **Jinja2**. U monitorovací části automatizovaného systému bude využívat nástroj **TIG stack**. Automatizační nástroje Ansible a Nornir budou použity pro sběr dat ze síťových zařízení a jejich uložení do **InfluxDB**. V případě Ubuntu serveru bude sbírat a ukládat data Telegraf agent.

3.3 Výběr nástrojů a obrazů emulovaných zařízení pro testování řešení

Obě řešení budou testována na síťové topologii, která bude vytvořena v grafickém síťovém simulátoru GNS3 (Graphical Network Simulator).

Řešení budou testována na následujících emulovaných zařízeních:

- **Cisco c7200** - Cisco router, operační systém Cisco IOS verze 15.2
- **Cisco IOSvL2** - Cisco L3 switch, operační systém Cisco IOS verze 15.2
- **Juniper JunOS Olive** - Juniper router, operační systém Junos OS verze 12.1R1.9
- **Ubuntu server** - server, operační systém Ubuntu 18.04, využití virtualizačního nástroje QEMU (Quick EMUlator).

Jelikož obě řešení budou testovány na obrazech, které jsou určeny pro testovací účely (neoficiální, nepodporované), tak Jinja2 šablony budou vytvářeny pro každé zařízení zvlášť.

3.4 Testovaná síťová topologie

Síťová topologie, na které budou testovány obě implementace automatizovaného systému, je uvedena na **obrázku č. 40**. V praktické části bude testována konfigurace v IPv4 i IPv6 prostředí. Jelikož v GNS3 je pro komunikaci mezi fyzickým počítačem a emulovanými zařízeními použita modifikace NAT node, tak management síť bude konfigurována pouze v IPv4 prostředí. Konfigurace FTPS serveru a TIG stacku bude provedena taky pouze v IPv4 prostředí.

Síťová topologie je tvořena následujícími zařízeními:

- **R1, R3** - Emulované Cisco směrovače, které využívají obraz Cisco c7200. Na zařízení R1 bude pomocí implementovaného systému provedena konfigurace síťových rozhraní (v IPv4 i IPv6 prostředí), OSPFv2 (Open Shortest Path First version 2), OSPFv3 (Open Shortest Path First version 3), backup, delete a restore konfigurace. V případě routeru R3 bude provedena konfigurace síťových rozhraní (v IPv4 i IPv6 prostředí), OSPFv2 i OSPFv3 (u obou konfigurací bude provedena redistribuce směrovacích informací z protokolu EIGRP). Dále bude na R3 provedena konfigurace EIGRP (IPv4 i IPv6 včetně redistribuce z protokolu OSPF). V neposlední řadě bude proveden backup, delete a restore konfigurace.
- **R2** - Emulovaný Juniper router, který používá obraz Juniper JunOS Olive. Na tomto zařízení bude pomocí nástrojů provedena konfigurace síťových rozhraní (v IPv4 i IPv6 prostředí), konfigurace OSPFv2 i OSPFv3, backup, delete a restore konfigurace.
- **MLS1** - Emulovaný Cisco L3 switch, který používá obraz Cisco IOSvL2. Cisco IOSvL2 nepodporuje OSPFv3. Na tomto zařízení bude pomocí nástrojů provedena konfigurace

VLAN, switching portů a SVI (v IPv4 i IPv6 prostředí). Konfigurovány budou VLAN 30, VLAN 40 a VLAN 50. Dále bude provedena konfigurace EIGRP (pro IPv4 i IPv6 prostředí), tak aby zařízení mohly mezi sebou komunikovat. Mimo jiné bude na zařízení konfigurováno ACL (Access-control list), tak aby zařízení ve VLAN 30 nemohly komunikovat se zařízeními ve VLAN 40 a naopak. V neposlední řadě bude otestován backup, delete a restore konfigurace.

- **Server1** - Server s operačním systémem Ubuntu 18.04. Na tomto serveru bude odzkoušena automatizovaná konfigurace FTPS serveru (v IPv4 i IPv6 prostředí) a TIG stacku (pouze v IPv4 prostředí). FTPS server i TIG stack bude potom testován pomocí PC_mgmt.
- **PC_mgmt** - GNS3 NAT node, který reprezentuje fyzický počítač s operačním systémem Ubuntu 18.04. Z tohoto počítače jsou spuštěny jednotlivé skripty/playbooky. Pro komunikaci PC_mgmt s ostatními síťovými prvky v GNS3 bylo nutné vytvořit virtuální síť **mgmt_network** pomocí nástrojů libvirt a virsh (viz sekce 3.5). Pro samotnou komunikaci je využito síťové rozhraní typu bridge (**virbr1_m**) s IP adresou **10.10.10.1**. PC_mgmt bude taky sloužit pro testování monitorovacího řešení a FTPS serveru.
- **PC1, PC2, PC3** - Docker kontejnery (Ubuntu Docker guests), které jsou používány pro testování konektivity (ping test). V podstatě se jedná o Docker kontejnery, které využívají upravený operační systém Ubuntu jako základní obraz (base image).
- **SW1, switch_mgmt** - Ethernet switche, které jsou použity pouze pro propojení některých prvků (např. v management síti). Tyto switche nejsou konfigurovány pomocí automatizačních nástrojů.

Po provedené automatizované konfiguraci by měly mezi sebou úspěšně komunikovat zařízení PC1, PC2, PC3 a Server1 vyjma zařízení PC2 a PC3, jelikož komunikace mezi VLAN30 a VLAN40 bude pomocí vytvořeného IP ACL zakázána. Management síť nebude distribuována protokoly EIGRP a OSPF. V testované síťové topologie bude dále odzkoušen sběr dat ze síťových zařízení a jejich výpis do konzole. Mimo jiné bude otestována možnost exportů dat (směrovací tabulky, paketové filtry) a vytváření reportů (základní údaje o síťových zařízeních, statistika příchozích a odchozích paketů).

3.5 Počáteční konfigurace síťových prvků

Při automatizování síťových zařízení je nejdříve nutné provést prvotní konfigurací síťových zařízení. Jelikož bude pro komunikaci mezí zařízeními použit protokol SSH, tak je nutné na jednotlivých zařízeních SSH nakonfigurovat. Mimo jiné je nutné na fyzickém počítači vytvořit virtuální síť, která bude mít roli management sítě. Dále bude nutné nakonfigurovat síťové rozhraní, tak aby bylo možné pomocí playbooků a skriptů vzdáleně spravovat zařízení, která jsou v management síti. **Všechny provedené počáteční konfigurace (vyjma PC_mgmt) jsou k dispozici**

v příloze C.

Na **PC_mgmt** je nejdříve nutné vytvořit management síť. Parametry sítě jsou popsány ve vytvořeném XML souboru **mgmt.xml** (viz výpis č. 2). Management síť s názvem mgmt_network využívá NAT pro překlad adres, tak aby například Server1 mohl mít přístup do Internetu v případě, že bude nutné stáhnout z repozitáře balíčky. Dále bude na PC_mgmt vytvořeno bridge rozhraní virbr1_m s IP adresou 10.10.10.1 a maskou 255.255.255.0, které bude použito v GNS3 (viz obrázek č. 40).

```
<network>
   <name>mgmt_network</name>
   <forward mode='nat'>
      <nat>
         <port start='1024' end='65535'/>
      </nat>
   </forward>
   <bridge name='virbr1_m' stp='on' delay='0'/>
   <ip address='10.10.10.1' netmask='255.255.255.0'></ip>
</network>
```

Výpis 2: Obsah souboru mgmt.xml

Po vytvoření souboru mgmt.xml a definování parametrů management sítě je nutné přidat vytvořenou virtuální síť do libvirt. Dále je nutné provést inicializaci sítě a povolit automatické spouštění virtuální síť mgmt_network v momentě, kdy je spuštěna služba libvirtd.

```
//Přidání nové virtuální sítě do libvirt.
$ virsh net-define mgmt.xml

//Inicializace virtuální síťe mgmt_network a povolení automatického spouštění
$ virsh net-start mgmt_network
$ virsh net-autostart mgmt_network
```

Na Cisco zařízeních (R1, R3, MLS1) je nutné nakonfigurovat SSH a síťové rozhraní pro správu zařízení (viz výpis č. 11, 13 a 14). Dále je povoleno směrování IPv6 unicast provozu a zakázán model AAA (Authentication, Authorization and Accounting) . U Cisco zařízeních je dále nutné povolit funkcionalitu SCP (Secure copy protocol) serveru, která je nutná při využívání NAPALM operací merge a replace. Dále je nutné nastavit archivaci konfigurace pomocí příkazu archive v případě, že je používána NAPALM operace replace. V případě MLS1 je nutné vypnout funkci auto-negotiation a nastavit duplex full manuálně. Pokud je funkce auto-negotiation zapnutá, tak nastane chyba duplex mismatch. Dále je nutné u MLS1 zakázat zobrazení bannerů v běžící konfiguraci, jelikož při nahrazení konfigurace pomocí NAPALM replace operace dojde

vždy k rollbacku staré konfigurace. V neposlední řadě je nutné provést konfiguraci VTP (VLAN Trunking Protocol) v transparentním režimu, tak aby se konfigurace VLAN zobrazila na výstupu příkazu show running-config. Pokud uživatel konfiguruje pomocí NAPALM knihovny zařízení s operačním systémem Cisco IOS, tak je důležité, aby si přečetl poznámky a prerekvizity, které je nutné brát v potaz při použití **ios ovladače** NAPALM knihovny.[1]

Na Juniper zařízení (R2) stačí provést konfiguraci SSH a síťového rozhraní (viz výpis č. 12). **Na PC1, PC2 a PC3 (Ubuntu Docker guests)** je provedena konfigurace síťového rozhraní eth0 (přiřazení IP adresy a defaultní brány pro IPv4 i IPv6 prostředí). Konfigurace PC1, PC2 a PC3 jsou uvedeny ve výpisech č. 16, 17 a 18. Konfigurace se přímo provádí v GNS3 kliknutím na uzel a zvolením možnosti **Edit config**. **Na zařízení Server1** je nutné nastavit síťová rozhraní pomocí nástroje Netplan. Nejdříve je nutné zakázat defaultní konfiguraci netplanu vytvořením souboru **99-disable-network-config.cfg**, který se musí nacházet v **/etc/cloud/cloud.cfg.d/**.
Obsah souboru 99-disable-network-config.cfg:

```
network: {config: disabled}
```

Následně byl vytvořen nový konfigurační soubor **01-netcfg.yaml**, který se nachází v **/etc/netplan/**. Obsah konfiguračního souboru je uveden pro Server1 ve výpisu č. 15. Na konec byly provedené změny aplikovány v netplanu následujícím příkazem:

```
sudo netplan apply
```

[1] https://napalm.readthedocs.io/en/latest/support/ios.html

4 Návrh automatizovaného systému pro konfiguraci a monitorování síťových zařízení pomocí nástroje Ansible

Vytvořený projekt **network_automation_ansible** je projekt, který využívá automatizační nástroj **Ansible verze 2.10.5**. Pro používání projektu **network_automation_ansible** je nutné mít nainstalovaný **Python interpreter verze 3.8 a více**. Mimo jiné pro konfiguraci TIG stacku je nutné mít nainstalované Ansible kolekce community.grafana a community.general. Stažení kolekcí bylo provedeno následujícím způsobem:

```
$ ansible-galaxy collection install community.grafana
$ ansible-galaxy collection install community.general
```

4.1 Správa Ansible projektu

Pro správu verzí jednotlivých souborů projektu je použit decentralizovaný verzovací systém Git. Obsah celého projektu je zveřejněn na webové platformě GitLab.[2] V Ansible projektu jsou jednotlivé Python balíčky včetně jejich závislostí spravovány pomocí nástroje Pipenv. Jednotlivé kroky pro práci s nástrojem Pipenv jsou stručně popsány ve výpisu č. 3. Pro vytvoření virtuálního prostředí je nutné zadat do příkazové řádky příkaz **pipenv install --three**. Mimo jiné se uživatel musí nacházat v kořenovém adresáři projektu. Seznam Python balíčků je obsažený v tzv. Pipfile souboru (viz výpis č. 19). V tomto souboru jsou uvedeny všechny nainstalované balíčky, které byly nainstalovány pomocí příkazu **pipenv install**. Následně je nutné vygenerovat tzv. **Pipfile.lock** soubor, který na základě Pipfile souboru specifikuje verze jednotlivých balíčků včetně jejich závislostí. Ve vytvořeném virtuálním prostředí lze hlavní skript (main.py) spustit například pomocí příkazu **pipenv run python main.py**. Pomocí **pipenv install** lze po naklonování projektu z GitLabu nainstalovat všechny Python balíčky z Pipfile souboru.

```
//Instalace pipenv balíčku pomocí správce balíčků pip
$ pip install pipenv

//Vytvoření virtuálního prostředí včetně Pipfile souboru
$ pipenv install --three

//Příklad instalace ansible balíčku pomocí Pipenv nástroje
$ pipenv install ansible

//Vygenerování Pipfile.lock souboru
$ pipenv lock
```

[2] https://gitlab.com/macaktom/network_automation_ansible

```
//Spouštění skriptu main.py ve virtuálním prostředí
$ pipenv run python main.py

//Instalace všech Python balíčků v Pipfile souboru - (při klonování projektu)
$ pipenv install
```

Výpis 3: Práce s Pipenv nástrojem

4.2 Struktura vytvořeného Ansible projektu

Výsledný Ansible projekt je složený z mnoha částí (playbooky, šablony, konfigurace atd.). V tomto oddílu je popsána základní struktura vytvořeného Ansible projektu. Struktura projektu je znázorněna na obrázku č. 41.

Základní popis struktury projektu:

- **collections** - Složka, ve které jsou uloženy kolekce, které byly vytvořené třetí stranou.
- **files** - Složka, která obsahuje podsložky s výstupními soubory.
- **files/backups** - Složka, ve které jsou uloženy zálohy běžící konfigurace jednotlivých síťových zařízení.
- **files/export** - Složka, která obsahuje podsložky, pod kterými jsou uloženy exporty obsahu síťové konfigurace, směrovacích tabulek a paketových filtrů. Dále jsou zde uloženy i jednotlivé reporty (HTML soubory).
- **files/staging** - Složka, do které jsou dočasně uloženy renderované Jinja2 šablony.
- **initial_setup** - Složka obsahující příkazy, které je nutné provést na jednotlivých zařízení před samotnou automatizaci pomocí nástroje Nornir (např. nastavení doménového jména, konfigurace SSH atd.).
- **inventory** - Obsahuje host.ini ve kterém jsou definovány jednotlivé skupiny a zařízení. V **podsložce examples** jsou příklady inventářů, kde jsou popsány jednotlivé parametry (pro každé zařízení zvlášť). Jednotlivé soubory tedy slouží pouze jako dokumentace pro uživatele. V **podsložce host_vars** lze nalézt soubory, ve kterých jsou definovány parametry konfigurace pro každé zařízení zvlášť. V **podsložce group_vars** lze nalézt podsložky skupin, které obsahují soubory s parametry dané skupiny a vault souborem pro uložení citlivých dat. V podsložce **vault** je uložen soubor **.vault_pass**, který obsahuje vault heslo.
- **log** - Obsahuje soubor ansible.log, který zaznamenává údálosti z provedených tasků.

- **templates** - Obsahuje Jinja2 šablony, které slouží pro konfiguraci daných zařízení. Šablony jsou vytvářeny podle výrobce zařízení (Cisco, Juniper, Debian), případně dle typu zařízení (router, L3_switch, server). Pouze šablony pro vytváření reportů nejsou umístěny v žádné podsložce adresáře **templates**.
- **ansible.cfg** - Konfigurační soubor, který slouží například k nastavení cesty k Python interpreteru, NAPALM pluginu, inventáři, kolekcím a k souboru s vault heslem.
- **Pipfile** - Definuje všechny použité knihovny pro development (včetně jejich závislostí).
- **db_playbook_runner.sh** - Bash skript, který spouští playbook db_handler.yml, který lze použít pro pravidelný sběr dat ze síťových zařízení a jejich následný zápis do InfluxDB databáze.
- ***.yml** - Spustitelné playbooky, které lze použít pro konfiguraci síťových zařízení a serverů, sběr dat, exportování dat atd.

4.3 Nastavení Ansible projektu

Pro nastavení Ansible projektu byl vytvořen konfigurační soubor **ansible.cfg** (viz obrázek č. 14), který se nachází v kořenovém adresáři projektu. V Ansible projektu je pomocí nástroje Pipenv nainstalována knihovna **napalm-ansible** [3], která obsahuje kolekci modulů, které využívají knihovnu NAPALM. V konfiguračním souboru **ansible.cfg** je nutné uvést cestu jak ke stažené knihovně (**library**), tak i k dodatečným pluginům (**actions_plugins**). V ansible.cfg bylo nutné nastavit cestu k inventáři (hosts.ini), logovacímu souboru ansible.log a souboru .vault_pass. Dále je zde uvedena cesta ke staženým kolekcím. V konfiguračním souboru lze pomocí direktiva **forks** nastavit počet vláken použitých pro provedení určitého Ansible úkolu. Pro monitorovací část systému je nutné povolit nativní datové typy u Jinja2 konstrukcí (**jinja2_native=True**).

```
[defaults]
library = ~/.local/share/virtualenvs/network_automation_ansible-jyRQ_HBH/lib/python3.8/site-packages/napalm_ansible/modules
action_plugins = ~/.local/share/virtualenvs/network_automation_ansible-jyRQ_HBH/lib/python3.8/site-packages/napalm_ansible/plugins/action
inventory = ./inventory/hosts.ini
module_utils = ~/.ansible/plugins/module_utils:/usr/share/ansible/plugins/module_utils
log_path = ./log/ansible.log
forks = 20
stdout_callback = yaml
vault_password_file = ./inventory/vault/.vault_pass
collections_paths = ./collections

#Povolení nativních datových typů u Jinja2 konstrukcí
jinja2_native=True
```

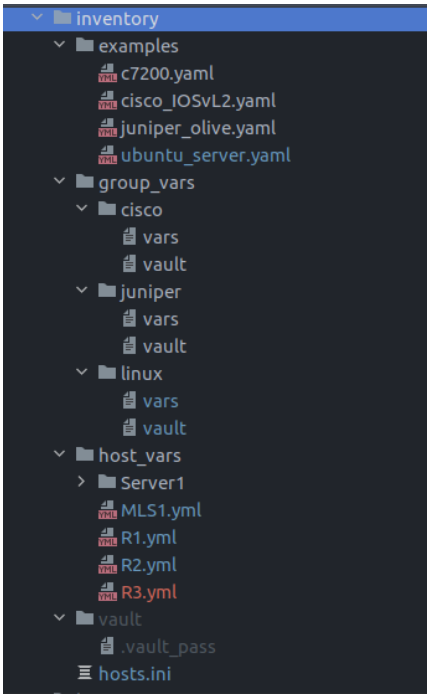
Obrázek 14: Konfigurační soubor ansible.cfg

[3] https://github.com/napalm-automation/napalm-ansible

4.4 Tvorba inventáře a zabezpečení citlivých dat

Inventář slouží k definování hostů, skupin a parametrů, které náleží jednotlivým hostům a skupinám. Vytvořené inventáře lze nalézt v adresáři inventory (viz obrázek č. 15). **Ve složce examples je uvedena dokumentace k jednotlivým atributům inventáře pro každý typ zařízení. Jsou zde uvedeny všechny atributy včetně způsobu, jak je použít.** Inventář byl rozdělen tak, aby v základním **hosts.ini** souboru byli uvedeni jednotlivý hosté a skupiny. Například host **MLS1** patří do skupiny **cisco_l3_switches**. Skupina **cisco_l3_switches** dále patří do skupiny **cisco_l3**, která obsahuje i skupinu **cisco_routers**. Aby skupina **cisco_l3** mohla obsahovat další skupiny, tak je nutné přidat k této skupině suffix **:children**. Skupina **cisco** obsahuje skupinu **cisco_l3**. Ostatní zařízení a skupiny byly definovány podobným způsobem. **Celý obsah souboru hosts.ini je uveden ve výpisu č. 21**.

Dále je nutné definovat parametry pro každý cílový uzel. Jednotlivé proměnné budou potom použity v Jinja2 šablonách nebo v Ansible úkolech. Proměnné a jejich hodnoty jsou definovány pro každé zařízení zvlášť. Vytvořené soubory jsou uložené ve složce **host_vars**. Jednotlivé proměnné jsou definovány dle specifikovaných požadavků (viz oddíl č. 3.4). Nejdříve byl u všech hostů definována proměnná ansible_host, která obsahuje IP adresu hosta pro management zařízení. Dále byly definovány proměnné jako vendor, dev_type (typ zařízení) a image (obraz). Tyto proměnné jsou použity například v podmínkách, na zakladě kterých se například provede Ansible úkol jen pro zařízení konkrétního vendora nebo typu.



Obrázek 15: Struktura adresáře inventory

4.4.1 Definování parametrů pro jednotlivé Cisco zařízení

Proměnné jednotlivých Cisco zařízeních jsou definovány v souborech **R1.yml (viz výpis č. 22), R3.yml (viz výpis č. 23) a MLS1.yml (viz výpis č. 24)**. V inventáři Cisco zařízení jsou definovány proměnné **interfaces_ipv4** a **interfaces_ipv6** pro konfiguraci síťových rozhraní v IPv4 i IPv6 prostředí. Uživatel specifikuje vždy o jaké rozhraní se jedná (např. FastEthernet0/0), a dále poskytne informace o tomto rozhraní (IP adresa, maska sítě, popis, duplex, speed, virtuální x fyzický port). V případě definování SVI u MLS1 lze specifikovat pouze popis, IP adresu a masku sítě. V případě proměnné **interfaces_ipv6** lze nastavit danému síťovému rozhraní i více IPv6 adres. Mimo jiné má uživatel možnost zvolit, jestli se má použít metoda EUI-64 (Extended Unique Identifier 64) pro vytvoření identifikátoru rozhraní. V MLS1 byly dále vytvořeny proměnné **vlans_config** a **switching_interfaces**, které se používají pro konfiguraci switching portů. Dále byly vytvořeny proměnné **ospf_config** a **ospfv3_config**, které jsou použity pro konfiguraci OSPFv2 a OSPFv3. Jelikož MLS1 nepodporuje OSPFv3, tak byla přidána možnost konfigurace EIGRP (pro IPv4 i IPv6 prostředí) pomocí proměnných **eigrp_config** a **eigrp_ipv6_config**. V případě **eigrp_config** je specifikováno číslo autonomního systému. Dále jsou v atributu **networks** definovány přímo připojené sítě. Naopak u **eigrp_ipv6_config** je nutné specifikovat číslo autonomního systému a názvy jednotlivých rozhraní (**proměnná routed_interfaces**), které budou používat EIGRP. V případě MLS1 si lze všimnout i proměnných **packet_filter_config** a **packet_filter_ipv6_config**, které se používají pro vytváření ACL. Vytvořeným klíčem (např. VLAN30-in) uživatel definuje název access listu. Packet_filter_config obsahuje klíč s názvem **rules**, ve kterém jsou definována jednotlivá pravidla, která spadají pod daný access list. Pomocí **routed_interfaces** lze definovat rozhraní, na které budou daná pravidla aplikována. V inventáři jsou dále uvedeny proměnné **restore_config** a **delete_config**. Proměnná restore_config se používá pro obnovení zálohované konfigurace specifikováním data provedené zálohy (viz hodnota klíče **running_config_date**). Proměnná **delete_config** se využívá pro mazání konfigurace. Uživatel zde může definovat části konfigurace, které se mají smazat z běžící konfigurace. Ostatní proměnné jsou uvedeny v dokumentaci ve složce examples.

4.4.2 Definování parametrů pro jednotlivé Juniper zařízení

Proměnné pro Juniper zařízení R2 jsou definovány v souboru R2.yml (viz výpis č. 25). Proměnné jsou definovány podobně jako u Cisco inventáře. Proměnné **interfaces_ipv4** a **interfaces_ipv6** slouží k definování síťových rozhraní. Mimo jiné uživatel má možnost definovat i logická rozhraní (units) daného fyzického rozhraní. Proměnné **ospf_config** a **ospfv3_config** slouží ke konfiguraci OSPFv2 a OSPFv3. Uživatel zde musí specifikovat router_id a jednotlivé rozhraní, která budou použita pro OSPF. Dále je v inventáři deklarovaná proměnná **restore_config**, která se používá k výběru zálohované konfigurace, která nahradí běžící konfiguraci zařízení. Proměnná **delete_config** slouží k mazání části konfigurace. Smazat lze například konfiguraci OSPF, fyzického nebo konkrétního logického rozhraní.

4.4.3 Definování parametrů pro Ubuntu server

Parametry pro **Server1** jsou rozděleny do dvou souborů: **vars** a **vault**. Ve **vars** souboru (viz výpis č. 26) je definována většina proměnných. Soubor **vault** obsahuje pouze citlivá data (hesla). Momentálně vault soubor obsahuje proměnnou **vault_vsftpd_user_password**. Tato proměnná je potom pomocí nástroje Ansible Vault a vault hesla zašifrována. Proměnná s prefixem vault je potom referencována pomocí Jinja2 výrazu v souboru vars. Součástí vars souboru zařízení Server1 je proměnná **vsftpd_config**, která se používá pro nastavení testovacího FTP(S) serveru (hodnota klíče **ssl**) a testovacího uživatele (hodnota klíče **vsftpd_user**). Dále jsou zde nastaveny parametry pro konfiguraci TIG stacku. Jsou zde definovány IP adresy, na kterých jsou jednotlivé služby provozovány. Dále jsou definovány proměnné **influx_db_url** a **grafana_url**, které reprezentují URL (Uniform Resource Locator) adresu pro přístup k jednotlivým službám. Dále je zde specifikována proměnná **grafana_data_sources**, která je použita při definování datových zdrojů (datasources) pro projekt Grafana. V rámci praktické části jsou definovány tři datové zdroje (influxsourceAnsible, influxsourceNornir, influxsourceTelegraf).

4.4.4 Definování parametrů jednotlivých skupiny

Parametry byly definovány pro tři skupiny: **cisco** (viz výpis č. 27), **juniper** (viz výpis č. 28) a **linux** (viz výpis č. 29). Pro každou skupinu je vytvořený samostatný adresař ve složce **group_vars**. Každý adresář obsahuje soubory **vars** a **vault**. Ve **vars** souboru je definována většina proměnných. Soubor **vault** obsahuje pouze citlivá data (hesla). Ve složce **vars** jsou u všech skupin definovány proměnné **ansible_user** a **ansible_password**, které se používají pro vzdálené připojení k jednotlivým zařízením. Samotné heslo je referencováno proměnnou **vault_ansible_password**, která byla deklarována ve vault souborech. Ve skupině linux byla mimo jiné specifikována proměnná **ansible_become_pass**, která slouží k eskalaci přístupových práv. Hodnota přístupového hesla je uloženo v souboru **vault**. Ve skupinách **cisco** a **juniper** je důležitá proměnná **napalm_provider**, která obsahuje slovník, který je používán modulem **napalm_ansible**. Je zde specifikován hostname, uživatelské jméno, heslo, typ operačního systému a dodatečné argument, mezi které můžeme zařadit **secret** a **global_delay_factor**. Pomocí argumentu global_delay_factor lze regulovat dobu čekání na výsledek z prováděného příkazu. Defaultně (global_delay_factor: 1) je nastavena maximální čekací doba 100s. V případě skupiny linux jsou definovány názvy jednotlivých InfluxDB databází, které budou vytvořeny v rámci konfigurace nástroje TIG stack.

4.4.5 Zabezpečení citlivých dat

Hesla a citlivé data jednotlivých skupin (cisco, juniper, linux) a zařízení Server1 jsou uložena v samostatných vault souborech. Tyto soubory obsahují proměnné, které mají prefix **vault**. Pomocí funkce Ansible Vault a vault hesla, které je uložené v souboru **.vault_pass**, jsou jednotlivé vault soubory zašifrovány a proměnné s **prefixem vault** jsou ve **vars** souborech referencovány

pomocí Jinja2 výrazů. Pro zabezpečení citlivých dat je nejdříve nutné definovat vault soubor s citlivými údaji:

```
---
vault_ansible_password: automationDP
vault_secret: cisco
```

Dále bylo nutné definovat vault heslo, které se použije pro zašifrování obsahu vault souboru. Heslo (**nornirdp**) je uloženo v souboru **.vault_pass** v adresáři vault. Pro šifrování obsahu vault souboru je použit nástroj **Ansible Vault**, který má definovaný příkaz **ansible-vault encrypt** pro zašifrování obsahu celého souboru. V tomto příkazu je nutné definovat soubor, který chceme zašifrovat. Dále byl využit volitelný argument **vault-password-file**, pomocí kterého byla specifikována lokace souboru s vault heslem (.vault_pass).

```
//Nutno provést v kořenovém adresáři Ansible projektu

//Příklad příkazu pro zašifrování vault souboru pro skupinu cisco
$ ansible-vault encrypt inventory/group_vars/cisco/vault \
  --vault-password-file inventory/vault/.vault_pass
```

Zašifrovaný vault soubor vypadá následovně (viz obrázek č. 16):

```
$ANSIBLE_VAULT;1.1;AES256
65366630353865386363313464653832316230336135656363626538343261323962643531663130
32636531616530363235656331646637636135613032383 20a6330623935323062373533633465 38
32333838396664356536333234353836356235313136303661306430373030643332306632383631
36663039373535313 60a3034323162613331356666316530386233353431303161623734663865 66
34383132663837383534303334353138363363346337636130373161343933393739656233613039
63626165386237353235656465303536623831656631313062 30
```

Obrázek 16: Zašifrovaný obsah souboru vault skupiny cisco

Jednotlivé proměnné s prefixem **vault** jsou potom referencovány v souboru **vars** (viz výpis č. 27). Pomocí Ansible Vault lze prohlížet nebo dešifrovat zašifrované soubory za předpokladu, že znáte vault heslo.

```
//Nutno se nacházet v kořenovém adresáři Ansible projektu

//Příklad příkazu na prohlížení vault souboru pro skupinu cisco
$ ansible-vault view inventory/group_vars/cisco/vault \
  --vault-password-file inventory/vault/.vault_pass

//Příklad příkazu na dešifrování vault souboru pro skupinu cisco
```

```
$ ansible-vault decrypt inventory/group_vars/cisco/vault \
  --vault-password-file inventory/vault/.vault_pass
```

4.5 Konfigurace síťových zařízení

Pro hromadnou konfiguraci síťových zařízení je nejdříve nutné definovat inventář včetně proměnných pro jednotlivé zařízení a skupiny (viz oddíl č. 4.4). Dále je nutné vytvořit pro každý typ zařízení Jinja2 šablonu, která obsahuje statické a dynamické části dané konfigurace (např. šablonu pro konfiguraci OSPFv3). V neposlední řadě je nutné vytvořit Ansible playbook, který obsahuje Ansible hru, pomocí které lze pro každé zařízení renderovat Jinja2 šablonu a sloučit renderovanou šablonu s běžící konfigurací zařízení.

4.5.1 Tvorba Jinja2 šablon

V rámci praktické části bylo vytvořeno hned několik šablon pro různé typy konfigurací dle požadavků, které byly na systém kladeny. Všechny dostupné šablony lze nalézt v adresáři **templates** (viz obrázek č. 43). Pokud tedy chceme například konfigurovat u jednotlivých zařízení síťová rozhraní v IPv4 prostředí, tak bude nutné vytvořit zvlášť pro každého vendora a každý typ zařízení (v inventáři proměnná **dev_type**) samostatnou šablonu se stejným názvem **interfaces_ipv4.j2**. To stejné platí i pro ostatní šablony vyjma souborů eigrp_ipv4, eigrp_ipv6, ospfv3.j2, switching_interfaces.j2, source_nat_overload.j2. Tyto šablony jsou vytvořené pouze pro některé typy zařízení. Jelikož v projektu je větší množství šablon, tak zde budou popsány pouze šablony pro konfiguraci OSPFv3 (soubor **ospfv3.j2**). Obsah souboru **ospfv3.j2** na konfiguraci OSPFv3 pro Cisco směrovače je uveden ve výpisu č. 30. NAPALM knihovna umožňuje u Cisco IOS zkontrolovat stav běžící konfigurace, ještě předtím než je renderovaná šablona sloučena s běžící konfigurací. Aby kontrola stavu konfigurace fungovala korektně, tak je nutné nejdříve zjistit, jak by daná konfigurace vypadala ve výstupu příkazu **show running-config**. Při vytváření šablon pro zařízení s operačním systémem Cisco IOS je nutné si dávat pozor na dodatečné mezery v šablonách. Mimo jiné v šabloně pro konfiguraci OSPFv3 je nutné nejen nakonfigurovat OSPFv3, ale provést konfiguraci i síťových rozhraní, které mají být využívány daným OSPFv3 procesem. Ve výstupu příkazu **show running-config** jsou tyto dvě konfigurace odděleny, proto je nutné i v samotné šabloně oddělit tyto konfigurace znakem **!**. U šablon pro operační systém Cisco IOS je nutné každou šablonu ukončit slovem **end**. Pokud toto není učiněno, tak může dojít k neočekávánému rollbacku konfigurace. Při renderování šablony je nejdříve provedena kontrola, jestli existuje v zparsovaném inventáři slovník ospfv3_config. Pokud existuje, tak je nejprve povoleno IPv6 unicast směrování.

```
ipv6 unicast-routing
!
```

Dále je provedena konfigurace OSPFv3 instance (procesu). Jelikož uživatel může v inventáři specifikovat pasivní rozhraní a možnosti redistribuce směrovacích informací, tak byly v šabloně vytvořeny Jinja2 konstrukce, které využívají proměnné passive_interfaces (list pasivních rozhraní) a redistribute (slovník údajů nutných pro redistribuci cest).

```
ipv6 router ospf {{ ospfv3_config.process }}
 router-id {{ ospfv3_config.router_id }}
{% if ospfv3_config.passive_interfaces is defined %}
{% for interface in ospfv3_config.passive_interfaces %}
 passive-interface {{ interface }}
{% endfor %}
{% endif %}
{% if ospfv3_config.redistribute is defined %}
{% if ospfv3_config.redistribute.static %}
 redistribute static
{% endif %}
{% if ospfv3_config.redistribute.eigrp is defined %}
{% for AS_number in ospfv3_config.redistribute.eigrp.AS_numbers %}
 redistribute eigrp {{ AS_number }}
{% endfor %}
{% endif %}
{% if ospfv3_config.redistribute.default %}
 default-information originate always
{% endif %}
{% endif %}
!
```

V neposlední řadě je nutné nakonfigurovaný OSPFv3 proces přiřadit síťovým rozhraním.

```
{% for int in ospfv3_config.interfaces %}
interface {{ int.name }}
 ipv6 ospf {{ ospfv3_config.process }} area {{ int.area_number }}
!
{% endfor %}
end
```

Vytváření Jinja2 šablon je podstatně jednodušší u Juniper zařízení. NAPALM knihovna srovnává renderovanou šablonu s výsledkem z příkazu **show configuration**. Jinja2 šablony lze pro Juniper zařízení vytvářet hierarchicky nebo pomocí **set příkazů**. V rámci této práce byla vybrána možnost tvorby šablon pomocí **set příkazů**, jelikož u tohoto způsobu není nutné se zabývat indentací a používáním složených závorek. Jinja2 šablona **ospfv3.j2** pro Juniper směrovače je

uvedena ve výpisu č. 31. Oproti šabloně pro Cisco IOS zde není potřeba řešit indentaci příkazů, oddělení jednotlivých konfigurací (znak !) a ukončení konfigurace (slovo end). Při renderování Jinja2 šablony pro konfiguraci OSPFv3 (viz výpis č. 31) je nejdříve provedena kontrola, jestli v inventáři má dané Juniper zařízení definovaný slovník **ospfv3_config**. Dále je provedeno nastavení **router id**. Potom jsou ve **for cyklu** procházeny uživatelem definovaná rozhraní, které budou využívány protokolem OSPFv3. Mimo jiné lze definovat i příznak **passive**, který konfiguruje dané fyzické rozhraní jako pasivní (v rámci protokolu OSPFv3). V neposlední je definována redistribuce statického směrování (volitelné). Uživatel musí specifikovat slovník **redistribute_static** včetně proměnných **policy_name** a **term_name**.

4.5.2 Tvorba Ansible scénářů

Jednotlivé Ansible playbooky (s příponou .yml) jsou vytvořeny v kořenovém adresáři projektu. Playbooky pro hromadnou konfiguraci síťových zařízení mají velmi podobnou implementaci, proto zde bude popsán pouze playbook **ospf_configuration.yml** pro konfiguraci OSPFv2 a OSPFv3 (viz výpis č. 36). Nejdříve je nutné se podívat na základní strukturu celého playbooku (viz obrázek č. 17).

```
//Vylistování všech Ansible her, včetně tasků a tagů
$ ansible-playbook ospf_configuration.yml --list-tasks
```

```
play #1 (routers): OSPFv2 configuration      TAGS: [ipv4]
  tasks:
    Create folders for parsed ospf configuration      TAGS: [ipv4]
    Set no conf commit variable (enable testing mode) TAGS: [ipv4]
    Parse ospf_ipv4 template  TAGS: [ipv4]
    Load and commit configuration     TAGS: [ipv4]
    Clean temporary folders/files     TAGS: [ipv4]
    Print differences in configs      TAGS: [ipv4]

play #2 (routers): OSPFv3 configuration      TAGS: [ipv6]
  tasks:
    Create folders for parsed ospf configuration      TAGS: [ipv6]
    Set no conf commit variable (enable testing mode) TAGS: [ipv6]
    Parse ospfv3 template     TAGS: [ipv6]
    Load and commit configuration     TAGS: [ipv6]
    Clean temporary folders/files     TAGS: [ipv6]
    Print differences in configs      TAGS: [ipv6]
```

Obrázek 17: Struktura playbooku ospf_configuration.yml

Každý implementovaný playbook pro hromadnou konfiguraci se skládá z několika Ansible her. Jelikož jeden z požadavku na Ansible projekt byl ten, aby šlo provádět konfiguraci pro prostředí IPv4 i IPv6, tak skoro každý Ansible scénář pro hromadnou konfiguraci síťových zařízení obsahuje minimálně 2 Ansible hry. Jedna z nich slouží ke konfiguraci v IPv4 prostředí a druhá potom v IPv6 prostředí. Obě tyto Ansible hry jsou vždy označkovány pomocí atributu tags, tak aby uživatel který provádí konfiguraci např. pouze v IPv6 prostředí, mohl vyfiltrovat pouze tu Ansible hru, kterou potřebuje. Z obrázku č. 17 a výpisu č. 36 lze vyčíst, že playbook ospf_configuration.yml se skládá ze dvou Ansible her: **OSPFv2 configuration** a **OSPFv3 configuration**. Každá tato hra je označena vlastním tagem. Ansible hra **OSPFv2 configuration** má definovaný **tag ipv4** a hra **OSPFv3 configuration** má v tags atributu definovaný **tag ipv6**. U obou her je v atributu hosts přiřazena skupina routers, jelikož dle testované topologie bude provedena konfigurace OSPFv2 a OSPFv3 na zařízeních R1, R2 a R3. Atribut **vars** obsahuje proměnnou, která je používána pro definování cesty k jednotlivým šablonám. Dále obsahuje proměnnou, která je později použita k vytváření složek, ve kterých jsou dočasně uloženy renderované šablony. V neposlední řadě je ve **vars** atributu definován příznak **commit_config**, který rozhoduje o tom, jestli bude nová konfigurace aplikována na běžící konfiguraci nebo jestli se uživateli pouze zobrazí rozdíly mezi konfiguracemi.

Popis Ansible úkolů v Ansible hře **OSPFv3 configuration** (viz obrázek č. 17 a výpis č. 36):

1. Nejdříve je provedena kontrola toho, jestli je v inventáři definován slovník ospfv3_config. Pokud není, tak se Ansible hra na daném zařízení neprovede.

2. Pomocí modulu **file** se provede kontrola, jestli existuje složka, která je uvedena v proměnné staging_ospf_folder. Pokud neexistuje, tak bude složka vytvořena. Provede se i pokud je zapnut testovací režim (check mode) při spuštění playbooku.

3. Provede se kontrola toho, jestli uživatel spustil playbook v testovacím módu. Podle toho se pomocí modulu **set_fact** nastaví příznak commit_config.

4. Pomocí modulu **template** se provede renderování vytvořené Jinja2 šablony do složky, která je uvedena v proměnné staging_ospf_folder. Soubor bude mít název podle názvu zařízení, které je uvedeno v hosts a přidané přípony .conf.

5. Pomocí modulu **napalm_install_config** a renderovaných Jinja2 šablon jsou konfigurována síťová zařízení. V argumentu **provider** je uveden slovník, který obsahuje údaje pro připojení k danému zařízení. Pomocí argumentu **config_file** je definována cesta k renderované šabloně. Pomocí příznaku **replace_config** lze specifikovat, jestli má nová konfigurace zcela nahradit běžící konfiguraci (operace replace) nebo se má nová konfigurace sloučit s běžící konfigurací (merge operace). V rámci konfigurace je použita operace merge (**replace_config: false**). Argument **commit_changes** určuje to, jestli se má nová kon-

figurace aplikovat. V případě testovacího režimu jsou do proměnné **result** zaznamenány pouze rozdíly v konfiguracích.

6. Dále jsou smazány pomocí **file** modulu všechny renderované šablony (včetně dočasné složky).

7. V neposlední řadě jsou pomocí **debug** modulu vypsány do konzole rozdíly mezi jednotlivými konfiguracemi daného zařízení (proměnná result).

Mimo jiné vytvořené Ansible úkoly jsou skoro identické v Ansible scénářích, které jsou určeny pro hromadnou konfiguraci síťových zařízení (např. **interfaces_configuration, eigrp_configuration** atd.). Liší se zejména hodnotou v hosts atributu. Dále se liší v názvech Jinja2 šablon a renderovaných šablon. Uživatel může při spuštění playbooku vyfiltrovat Ansible hry, které se musí provést. Dále uživatel může využít i implementovaného testovacího režimu (check mode).

Jednotlivé možnosti spouštění vytvořených Ansible playbooků jsou popsány v následujícím výpisu:

```
//Konfigurace OSPFv2 (filtrace Ansible her pomocí tagu ipv4)
$ ansible-playbook ospf_configuration.yml --tags ipv4

//Konfigurace OSPFv3 (filtrace Ansible her pomocí tagu ipv6)
$ ansible-playbook ospf_configuration.yml --tags ipv6

//Konfigurace OSPFv2 i OSPFv3
$ ansible-playbook ospf_configuration.yml

//Konfigurace OSPFv2 i OSPFv3 - testovací režim
$ ansible-playbook ospf_configuration.yml --check
```

4.5.3 Testování playbooků

V rámci testované topologie (viz oddíl 3.4) byly pro hromadnou konfiguraci síťových zařízení použity následující příkazy:

```
//Konfigurace síťových rozhraní
$ ansible-playbook interfaces_configuration.yml

//Konfigurace OSPF - skupina routers
$ ansible-playbook ospf_configuration.yml

//Konfigurace EIGRP - R3, MLS1
```

```
$ ansible-playbook eigrp_configuration.yml

//Konfigurace ACL - MLS1
$ ansible-playbook packet_filter_configuration.yml
```

Na obrázku č. 18 je zobrazen výstup po provedení Ansible hry **Switching interfaces configuration**. Na obrázku č. 18a je zobrazen výsledek konfigurace switching portů, která byla sloučena s běžící konfigurací zařízení MLS1. Po druhém spuštění Ansible hry **Switching interfaces configuration** (viz obrázek č. 18b) jsou na jednotlivých rozhraních provedeny pouze příkazy switchport a no shutdown. Vytvořená Ansible hra tedy není idempotentní, jelikož oba tyto příkazy se nenachází ve výstupu příkazu show running-config. Tyto příkazy však musí být uvedeny v šabloně, jelikož se při některých spuštěních dané hry stávalo, že některé příkazy nebyly vůbec provedeny. Chyba se může nacházet v samotném obrazu Cisco IOSvL2 nebo v implementaci NAPALM knihovny.

```
TASK [Print differences in configs] **********
ok: [MLS1] =>
  msg: |-
    changed: true
    failed: false
    msg: |-
        +vlan 30
        + name VLAN30
        +vlan 40
        + name VLAN40
        +vlan 50
        + name VLAN50
        +interface GigabitEthernet0/2
        + description connected to PC2 on port eth0
        + switchport
        + switchport access vlan 30
        - no shutdown
        +interface GigabitEthernet0/3
        + description connected to PC3 on port eth0
        + switchport
        + switchport access vlan 40
        - no shutdown
        +interface GigabitEthernet1/0
        + description connected to Server1 on port ens4
        + switchport
        + switchport access vlan 50
        - no shutdown
```

(a) Po prvním provedení Ansible hry

```
TASK [Print differences in configs] ***
ok: [MLS1] =>
  msg: |-
    changed: true
    failed: false
    msg: |-
        +interface GigabitEthernet0/2
        + switchport
        - no shutdown
        +interface GigabitEthernet0/3
        + switchport
        - no shutdown
        +interface GigabitEthernet1/0
        + switchport
        - no shutdown
```

(b) Po druhém provedení Ansible hry

Obrázek 18: Výpis rozdílů v konfiguracích po provedení Ansible hry **Switching interfaces configuration**

Na obrázku č. 44 je zobrazen výpis rozdílů mezi novou konfigurací a běžící konfigurací u jednotlivých zařízení po prvním a druhém provedení Ansible hry **OSPFv3 configuration**. Po druhém provedení hry **OSPFv3 configuration** je u všech zařízeních ve výpisu uveden příznak **changed** jako false. Ansible hra **OSPFv3 configuration** je tedy zcela idempotentní.

4.6 Sběr dat ze síťových zařízení

Pro sběr dat ze síťových zařízení a jejich následný výpis do konzole byl vytvořen playbook **network_info_viewer.yml**. Tento playbook je složen z patnácti Ansible her. Playbook obsahuje Ansible hry například pro výpis směrovacích tabulek, OSPF sousedů, IP adres jednotlivých síťových rozhraní, základních údajů o síťových zařízení atd. Pro sběr dat se využívají především NAPALM gettery [4], které vrací strukturované data. Pokud není pro potřebnou funkcionalitu implementován NAPALM getter, tak je nutné pro jednotlivé operační systémy (Cisco IOS, Junos OS) najít konkrétní příkazy, které by uživatel zadal pomocí CLI. Obdržená nestrukturovaná data je však nutné případně zparsovat pro potřeby dané Ansible hry. Příkladem jsou například příkazy **show ipv6 route** (Cisco IOS) a **show route** (Junos OS). Výstup příkazu show route zobrazuje IPv4 i IPv6 směrovací tabulku, přičemž **show ipv6 route** zobrazuje pouze IPv6 směrovací tabulku. Ansible hra **Show configured VLANs** (viz výpis č. 43) využívá NAPALM get_vlans. Pro spuštění této hry lze použít tag **vlan**. Ansible hra **Show configured VLANs** je tvořena ze dvou Ansible úkolů: **Get configured VLANs** a **Print configured VLANs**. Pro sběr dat je použit modul napalm_get_facts, který obsahuje argumenty **provider** a **filter**. Do argumentu provider bylo nutné referencovat vytvořený slovník **napalm_provider** (podobně jako u Ansible her pro konfiguraci síťových zařízení). Pomocí argumentu lze specifikovat jednu nebo více NAPALM getter funkcí, které se mají provést. Jelikož chceme získat informace o jednotlivých VLANách, tak do argumentu **filter** bylo nutné uvést filter **vlans** (NAPALM getter **get_vlan** bez prefixu get_). Výsledná data jsou potom přiřazena proměnné **result**. Obsah této proměnné je vypsán do konzole pomocí modulu debug. Ansible hru **Show configured VLANs** můžeme spustit následujícím způsobem:

```
//Zobrazit všechny Ansible hry a jejich tagy
$ ansible-playbook network_info_viewer.yml --list-tags

//Výpis konfigurovaných VLAN - zařízení MLS1
$ ansible-playbook network_info_viewer.yml --tags vlan
```

Jelikož na zařízení MLS1 byly pomocí playbooku interfaces_configuration.yml konfigurovány VLAN 30, VLAN 40 a VLAN 50, tak po provedení Ansible hry **Show configured VLANs** byly tyto VLANy vepsány do konzole (viz obrázek č. 45).

[4] https://napalm.readthedocs.io/en/latest/support/

Na druhou stranu Ansible hra **Show OSPFv3 neighbors** (viz výpis č. 44) nevyužívá NAPALM gettery. Pro spuštění hry je deklarován tag **ospfv3**. Dále v atributu **vars** je vytvořen slovník **commands**, ve kterém jsou uvedeny příkazy (show ospfv3 neighbor, show ipv6 ospf neighbor) pro zobrazení OSPFv3 sousedů podle vendora. V bloku jsou uvedeny dva Ansible úkoly, které se provedou pouze, pokud hodnota proměnné **vendor** daného hosta je rovna jednomu z klíčů slovníku commands (tzn. **cisco** nebo **juniper**). V takovém případě se provede nejdříve task **Get OSPFv3 neighbors**, který pomocí modulu **napalm_cli** provede na síťovém zařízení příkaz, který je specifikován v argumentu commands. Výsledná data jsou potom přiřazena proměnné **result**. Obsah této proměnné je vypsán do konzole pomocí modulu debug.

4.7 Exportování dat a tvorba reportů

Pro exportování dat a tvorbu reportů byl vytvořen playbook **network_info_exporter.yml** (viz výpis č. 37). Playbook je složen z šesti Ansible her. Jednotlivé hry včetně jejich tagů jsou uvedeny na obrázku č. 19. Exportovat lze například běžící konfiguraci, obsah směrovací tabulky nebo pravidla pro filtrování paketů. Export dat do souborů s příponou .conf nebo .txt je vždy proveden paralelně. Dále lze vytvářet i HTML reporty. První typ HTML reportu sumarizuje základní informace o síťových zařízení. Na druhou stranu druhý typ HTML reportu poskytuje základní statistiku o přijatých a vysílaných paketech na jednotlivých rozhraní. Při vytváření reportů jsou data nejdříve sbírána paralelně, vzápětí jsou však data jednotlivých zařízení agregována. Tyto data jsou potom využita při renderování jedné Jinja2 šablony. Po renderování Jinja2 šablony je vytvořen HTML soubor.

```
play #1 (network_devices): Export device configuration        TAGS: [run_conf]
    TASK TAGS: [run_conf]

play #2 (routers, cisco_l3_switches): Export IPv4 routes      TAGS: [ip_routes,ipv4_routes]
    TASK TAGS: [ip_routes, ipv4_routes]

play #3 (routers, cisco_l3_switches): Export IPv6 routes      TAGS: [ipv6_routes,ip_routes]
    TASK TAGS: [ip_routes, ipv6_routes]

play #4 (routers, cisco_l3_switches): Export packet filters info      TAGS: [acl,packet_filters]
    TASK TAGS: [acl, packet_filters]

play #5 (network_devices): Export packet counters data to HTML        TAGS: [packets_counter]
    TASK TAGS: [packets_counter]

play #6 (network_devices): Export devices facts to HTML       TAGS: [facts]
    TASK TAGS: [facts]
```

Obrázek 19: Struktura playbooku network_info_exporter.yml

4.7.1 Implementace Ansible her pro exportování dat

Pro exportování dat byly vytvořeny čtyři Ansible hry **Export device configuration, Export IPv4 routes, Export IPv6 routes a Export packet filters info**. Všechny tyto hry vyjma **Export device configuration** exportují specifická data jednotlivých zařízení do souboru s příponou .txt. U Ansible hry Export device configuration je běžící konfigurace exportována do souboru s příponou .conf. Jednotlivé hry fungují na podobném principu, pokud nebereme v potaz specifické parsování dat a složku, ve které jsou exportované soubory k dispozici. Obsah playbooku **network_info_exporter.yml** je uveden ve výpisu č. 37. Ve vybrané Ansible hře Export IPv6 routes jsou definovány jednotlivé příkazy pro zobrazení IPv6 směrovací tabulky. Dále jsou deklarovány proměnné **ipv6_pattern** a **ipv4 pattern**, které později slouží k oddělení obsahu směrovacích tabulek pro IPv4 a IPv6 prostředí. Ve hře jsou definovány dva tagy **ipv6_routes** a **ip_routes**. Pomocí tagu ip_routes se provede export IPv4 i IPv6 směrovací tabulky. Nejdříve jsou získány směrovací tabulky jednotlivých zařízení. Výsledek je uložen do proměnné result. Pokud neexistuje žádná IPv6 cesta, tak je proměnné result přiřazen prázdný string. Pokud IPv6 cesty existují v daném výstupu, tak u Juniper zařízení je nutné oddělit IPv4 a IPv6 cesty pomocí deklarovaných vzorů. Dále je vytvořena složka ip_routes (případně i složka export). V posledním tasku jsou paralelně exportována data do souboru s příponou .txt pomocí modulu **copy**. V tomto modulu je nutné v argumentu **content** specifikovat data, které chceme exportovat. V argumentu **dest** je nutné upřesnit cestu k samotnému souboru, která je v implementovaném tasku definována dynamicky podle hodnoty proměnné export_ip_routes_folder, názvu zařízení a statické části _ipv6.txt.

4.7.2 Implementace Ansible her pro tvorbu HTML reportů

Pro tvorbu sumarizačních HTML reportů byly vytvořeny dvě Ansible hry: **Export packet counters data to HTML** (s tagem packets_counter) a **Export devices facts to HTML** (s tagem facts), které jsou součástí playbooku **network_info_exporter.yml**, jehož obsah je uveden ve výpisu č. 37. V Ansible hře **Export packet counters data to HTML** je velmi důležitá proměnná **sorted_header**, která obsahuje seřazený list klíčů, který je později použit při seřazení dat v HTML tabulce Jinja2 šablony. Nejdříve jsou paralelně posbíraná data z jednotlivých zařízení pomocí **NAPALM getteru** get_interfaces_counters. Dále byly všechny výsledky agregovány do jednoho slovníku **aggregated_dict**, který je potom dále zpracován ve vytvořené Jinja2 šabloně. Dále se vytvoří složka export, pokud ještě nebyla vytvořena ve složce files. Na závěr je provedeno renderování Jinja2 šablony **packets_counter.j2** (viz výpis 32). V Jinja2 šabloně byla pro každé zařízení vytvořena tabulka, jejíž obsah je dynamicky vytvořen dle obdržených dat. Proměnná **all_hosts** obsahuje list, který je procházen v Jinja2 šabloně, tak aby byl uživatel při kliknutí na název daného zařízení odkázán na tabulku toho zařízení. Po renderování Jinja2 šablony je vytvořen HTML dokument, který obsahuje data ze slovníku **aggregated_dict**. Na výsledný HTML dokument jsou aplikovány kaskádové styly.

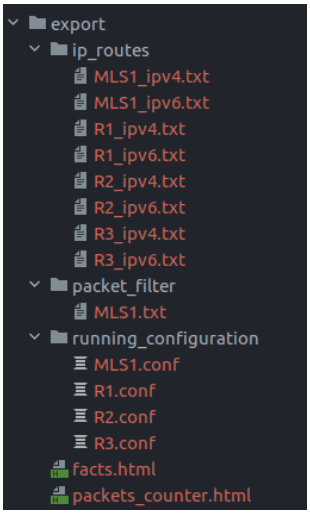
Ansible hra **Export devices facts to HTML** je součástí playbooku network_info_exporter.yml (viz výpis č. 37) . V Ansible hře **Export devices facts to HTML** bylo nutné definovat proměnnou sorted_header, která jako u minulé hry obsahuje seřazený list klíčů, který je později použit při seřazení dat v HTML tabulce Jinja2 šablony. Nejdříve jsou pomocí NAPALM getteru **get_facts** posbírány základní údaje o jednotlivých zařízeních. U Juniper Olive je nutné navíc přiřadit do proměnné result vendora, jenž je definován v inventáři daného hosta. Dále bylo nutné u Cisco zařízeních zparsovat verzi operačního systému, kterou lze najít ve slovníku **result** pod klíčem **os_version**. Verze operačního systému byla zparsovaná tak, aby byla pod klíčem **os_version** uložena pouze hodnota verze používaného Cisco IOS. Dále bylo nutné zparsovat hodnotu **uptime**, která defaulně je definována v sekundách. Uptime byl zparsován pro lepší čitelnost do časového formátu H:M:S. Jednotlivé výsledky byly potom agregovány do jednoho slovníku **aggregated_dict**, který je potom dále zpracován ve vytvořené Jinja2 šabloně. Dále je vytvořena složka exports, pokud ještě neexistuje. Poslední task renderuje šablonu **facts.j2** (viz výpis č. 33). Na rozdíl od šablony **packets_counter.j2** je zde vytvořena pouze jedna HTML tabulka, která bude obsahovat veškeré informace o jednotlivých zařízeních. Hlavička tabulky je vytvořena pomocí jednotlivých prvků listu **sorted_header**. Pro vytvoření těla tabulky bylo nutné pomocí for cyklu procházet klíči slovníku **aggregated_dict**. Každý klíč slovníku aggregated_dict tvoří další slovník, který obsahuje údaje o jednom konkrétním hostu. Pro seřazení dat bylo nutné taky procházet jednotlivé prvky listu sorted_header. Jednotlivé prvky tohoto listu byly použity jako klíče pro seřazení dat daného hosta. Po renderování Jinja2 šablony je vytvořen HTML dokument, který obsahuje data ze slovníku **aggregated_dict**. Na výsledný HTML dokument jsou taktéž aplikovány kaskádové styly.

4.7.3 Testovani playbooku network_info_exporter.yml

Pro vytvoření všech implementovaných exportů a reportů stačí pouze spustit samotný playbook. Pokud chceme využít pouze některé funkcionality (např. pouze tvorba reportů), tak je nutné spustit Ansible playbook pomocí daného tagu.

```
//Zobrazení jednotlivých Ansible her a jejich tagů
$ ansible-playbook network_info_exporter.yml --list-tags

//Exportování běžící konfigurace
$ ansible-playbook network_info_exporter.yml --tags run_conf

//Spuštění všech her (exportování i tvorba reportů)
$ ansible-playbook network_info_exporter.yml
```

Po provedení následujícího playbooku vznikne složka export, ve které jsou vytvořeny podsložky, které obsahují soubory s exportovanými daty (viz obrázek č. 20):



Obrázek 20: Struktura složky export

Po spuštění playbooku network_info_exporter.yml byly mimo jiné vytvořeny dva HTML reporty: facts.html (viz obrázek č. 21) a packets_counter.html (viz obrázek č. 22).

Devices facts export

hostname	FQDN	vendor	model	serial_number	os_version	uptime
R1	R1.automation.local	Cisco	7206VXR	4279256517	Version 15.2(4)S5	05:38:00
R3	R3.automation.local	Cisco	7206VXR	4279256517	Version 15.2(4)S5	00:09:00
MLS1	MLS1.automation.local	Cisco	IOSv	9LD1YQMD0KM	Version 15.2(4.0.55)E	00:09:00
R2	R2.automation.local	Juniper	OLIVE	-	12.1R1.9	05:38:39

Obrázek 21: HTML report facts.html

Packet counters tables

Devices:

- R1
- R3
- MLS1
- R2

R1

interface	rx_broadcast_packets	rx_discards	rx_errors	rx_multicast_packets	rx_octets	rx_unicast_packets	tx_discards	tx_errors	tx_octets	tx_unicast_packets
Ethernet1/0	0	0	0	0	1446206	15989	0	0	897808	7508
Ethernet1/1	0	0	0	0	0	0	0	0	0	0
Ethernet1/2	0	0	0	0	0	0	0	0	0	0
Ethernet1/3	0	0	0	0	0	0	0	0	0	0
Ethernet1/4	0	0	0	0	0	0	0	0	0	0
Ethernet1/5	0	0	0	0	0	0	0	0	0	0
Ethernet1/6	0	0	0	0	0	0	0	0	0	0
Ethernet1/7	0	0	0	0	0	0	0	0	0	0
FastEthernet0/0	4707	0	0	0	449870	4757	0	0	728646	7048
FastEthernet0/1	5	0	0	0	8848	96	0	0	308658	2617

Obrázek 22: HTML report packets_counter.html (zařízení R1)

4.8 Zálohování, obnovení a mazání konfigurace

Zálohování, obnovení a mazání konfigurace jsou velmi důležité operace, které je vhodné také automatizovat. Pro každou operaci byl vytvořen samostatný playbook. Pro zálohování běžící konfigurace byl vytvořen playbook **backup_configuration.yml** (viz výpis č. 38). Pro mazání jednotlivých částí konfigurace byl vytvořen playbook **delete_configuration.yml** (viz výpis č. 39). Pro nahrazení běžící konfigurace za zálohovanou konfigurace byl vytvořen playbook **restore_configuration.yml** (viz výpis č. 40)

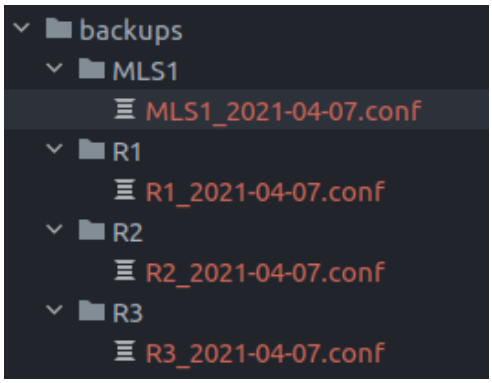
4.8.1 Zálohování běžící konfigurace

Playbook **backup_configuration.yml** (viz výpis č. 38) je složen z jedné Ansible hry s názvem **Backup network device configuration**. Nejdříve je z řídícího uzlu získán aktuální čas a datum pomocí příkazu **date** (ve formátu Y-m-d H-M-S). Výsledek toho tasku byl uložen do proměnné **timestamp**. Pomocí NAPALM getteru **get_config** byla získána startovací i běžící konfigurace jednotlivých zařízení a uložena do proměnné **result**. Dále byl implementovaný task pro vytvoření složky backup a podsložek, které jsou pojmenovány dle názvu zařízení. V neposlední řadě je nutné data exportovat do souboru s příponou .conf pomocí modulu **copy**. V argumentu **content** je referencován obsah běžící konfigurace. V argumentu **dest** je uveden název souboru včetně jeho umístění. Název souboru je odvozený z názvu zařízení, data provedené zálohy, které je ve formátu Y-m-d a přípony .conf.

Pro vytvoření zálohy běžící konfigurace stačí spustit vytvořený playbook následujícím způsobem:

```
//Zálohání běžící konfigurace
$ ansible-playbook backup_configuration.yml
```

Po provedení následujícího playbooku vznikne složka backups, kde lze nalézt jednotlivé zálohy (viz obrázek č. 23):



Obrázek 23: Struktura složky backups

4.8.2 Mazání konfigurace

Playbook **delete_configuration.yml** (viz výpis č. 39) je složen z jedné Ansible hry s názvem **Deletion of network devices configuration**. Implementace playbooku je velmi podobná implementaci ostatním playbookům pro hromadnou konfiguraci síťových zařízení. Playbook lze spouštět v testovacím režimu. Nejprve byly vytvořeny Jinja2 šablony **delete_configuration.j2**, ve kterém jsou specifikovány jednotlivé příkazy, které jsou použity pro smázání jednotlivých částí konfigurace (např. síťová rozhraní, OSPF, EIGRP atd.). V inventáři musí uživatel ve slovníku **delete_config** specifikovat, které části konfigurace budou později smazány. Pro Cisco zařízení jsou tyto šablony vytvořeny pomocí imperativního přístupu, jelikož šablony obsahují příkazy, které začínájí se slovem **no**. Co se týče implementace playbooku, tak nejdříve bylo nutné vytvořit task, který dokáže vytvořit podsložku **delete** ve složce **staging**, kde budou uloženy renderované šablony. Dále bylo nutné vytvořit task, který zkontroluje, jestli je playbook spuštěn v testovacím režimu. Na základě provedené kontroly bude nastaven příznak proměnné **commit_config**. Dále je provedeno renderování šablon **delete_configuration.j2** pomocí modulu **template**. Pomocí modulu **napalm_install_config** jsou na jednotlivých zařízeních provedeny příkazy, které jsou uvedeny v renderovaných šablonách. Jednotlivé příkazy jsou aplikovány pomocí NAPALM operace **merge** (replace_config: false). Po aplikování příkazů jsou dále pomocí tasku **Clean temporary folders/files** smazány všechny renderované šablony včetně samotné složky delete. V neposlední řadě byl proveden výpis hodnoty proměnné **result** do konzole.

Pokud chceme otestovat mazání konfigurace, tak je nutné nadefinovat v inventáři ve slovníku delete_config jednotlivé části konfigurace, které chceme smazat. Pro zařízení R1 byl slovník delete_config definován následovně:

```
delete_config:
   interfaces:
      physical: [FastEthernet0/0, FastEthernet0/1]
   ospf_config:
      processes: [1]
   ospfv3_config:
      processes:
         - number: 1
           interfaces:
            - name: FastEthernet0/0
              area_number: 0
            - name: FastEthernet0/1
              area_number: 0
```

Pokud chceme tedy u všech zařízení odstranit části konfigurace, které byly provedeny v oddílu č. 4.5.3, tak je nutné jednotlivé části konfigurace pro daná síťová zařízení definovat v delete_config

(viz příloha F). Potom stačí následujícím způsobem spustit playbook **delete_configuration.yml**:

```
//Mazání určitých částí konfigurace síťových zařízení
$ ansible-playbook delete_configuration.yml
```

4.8.3 Obnovení zálohované konfigurace

Obnovení konfigurace je implementováno, tak aby mohl uživatel nahradit běžící konfiguraci za zálohovanou konfiguraci dle specifikovaného data v inventáři daného hosta. Playbook **restore_configuration.yml** (viz výpis č. 40) je složen z jedné Ansible hry s názvem **Restore network device configuration**. Playbook lze spustit v testovacím módu. V takovém případě budou vypsány pouze rozdíly mezi běžící a vybranou zálohovanou konfigurací. Implementace playbooku je velmi podobná implementaci ostatních playbooků pro hromadnou konfiguraci síťových zařízení. Nejdříve je nastaven příznak **commit_config** na základě toho, jestli je playbook spuštěn v testovacím režimu (check mode). Potom pomocí **napalm_install_config** je provedeno samotné nahrazení konfigurace. Oproti jiným implementovaným playbookům využívá tento playbook NAPALM operaci **replace** (replace_config: true). V argumentu **config_file** je uvedena cesta k zálohovanému konfiguračnímu souboru. Pro výběr konfiguračního souboru (zálohy) je nutné v inventáři specifikovat slovník **restore_config** včetně hodnoty klíče running_config_date. Jako hodnota atributu running_config_date musí být uvedeno datum ve formátu Y-m-d.

Pro nahrazení běžící konfigurace za zálohovanou konfiguraci je nutné mít nejdříve vytvořenou nějakou zálohu (viz oddíl č. 4.8.1). V předchozím oddílu bylo provedeno smazání jednotlivých částí konfigurace. V inventáři jednotlivých zařízení je nutné definovat slovník restore_config. Jelikož záloha byla provedena 7. dubna 2021, tak je nutné v inventářích definovat slovník restore_config následovně:

```
restore_config:
   running_config_date: 2021-04-07
```

Playbook **restore_configuration.yml** je spuštěn následujícím způsobem:

```
//Obnovení zálohované konfigurace
$ ansible-playbook restore_configuration.yml
```

Po provedení Ansible hry je nahrazena běžící konfigurace za zálohovanou konfiguraci.

4.9 Konfigurace Ubuntu serverů

V rámci diplomové práce byla provedena implementace playbooku **linux_configuration.yml** (viz výpis č. 41), který slouží ke konfiguraci **FTPS serveru** a **TIG stacku** na serverech s

operačním systémem **Ubuntu 18.04**. Playbook je složen ze dvou Ansible her: **Setup and Configure FTPS server** a **Setup TIG Stack**. První Ansible hra slouží ke konfiguraci služby **vsftpd**. Uživatel může konfigurovat FTP nebo FTPS server. V případě konfigurace FTPS serveru je automaticky generován vlastní self-signed certifikát pomocí **knihovny OpenSSL**. V rámci hry je vytvořen i uživatel pro testování samotné služby. Druhou hru lze použít pro instalaci a konfiguraci nástroje TIG stack včetně vytvoření jednotlivých databází v InfluxDB a definování vytvořených datových zdrojů v projektu Grafana. V obou Ansible hrách je nutné nastavit privilegovaný režim pomocí atributu **become**.

4.9.1 Tvorba Jinja2 šablon

Pro servery s Ubuntu 18.04 byly vytvořeny dvě Jinja2 šablony: **vsftpd.j2** a **telegraf.j2**. Šablona **vsftpd.j2** (viz výpis č. 34) obsahuje konfigurace složbu vsftpd (FTP případně FTPS server). Šablona bude renderována v případě, že je definován slovník **vsftpd_config**. V základní konfiguraci je zejména provedeno nastavení práv pro jednotlivé operace, definování kořenové složky ftp, nastavení userlistu a povolení IPv6 protokolu. Pokud je definován slovník **ssl** (ve slovníku vsftpd_config) a zároveň je povolena konfigurace SSL (Secure Sockets Layer)/TLS (Transport Layer Security) ve slovníku **ssl** (enabled: true), tak renderovaná šablona bude obsahovat i dodatečnou SSL/TLS konfiguraci pro službu vsftpd. Pro nastavení SSL/TLS bylo nutné v šabloně povolit SSL, zakázat anonymní připojení a povolit TLS (ssl_tlsv1=YES), případně **lze** povolit i SSL (direktivum ssl_sslv3=YES). Cesta k privátnímu klíči a certifikátu je uložena v proměnných, které jsou definovány v inventáři (ve slovníku **ssl**).

Jinja2 šablona **telegraf.j2** (viz výpis č. 35) slouží k vytvoření konfiguračního souboru pro konfiguraci služby telegraf. V rámci šablony je deklarována konfigurace agenta, output a input pluginů. V sekci **agent** byl definován zejména interval a hostname. Telegraf agent sbírá a zasílá data každých 20s (interval = ¨20s¨). V případě **output pluginů** je použit plugin influxdb. U tohoto pluginu je pomocí Jinja2 výrazů referencován InfluxDB URL, název databáze (do které má služba telegraf ukládat data), uživatel a heslo. Jako **input pluginy** byly využity pluginy **cpu** a **system**. V případě **cpu** je nutné nastavit totalcpu = true, jelikož dle požadavků je nutné později zobrazit celkové zatížení procesoru a ne jednotlivých jader. Plugin **system** lze například použít pro zjištění průměrného zatížení systému nebo uptime hodnoty.

4.9.2 Konfigurace FTP(S) serveru

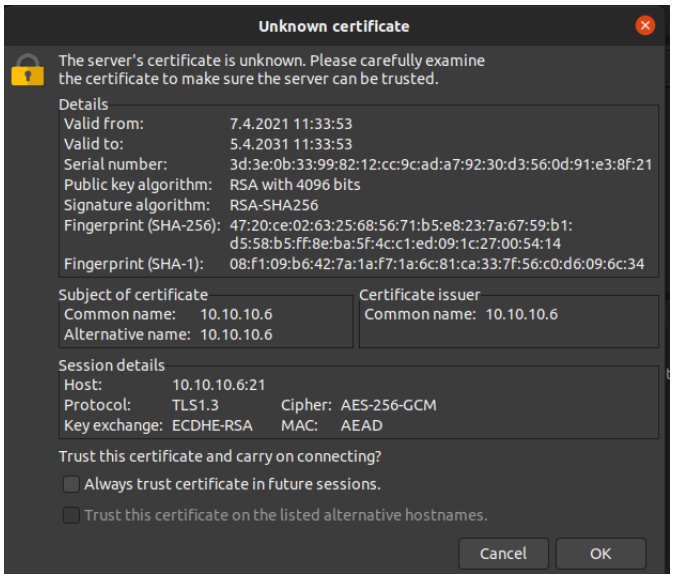
Hra **Setup and Configure FTPS server** je součástí playbooku **linux_configuration.yml** (viz výpis č. 41). Pomocí modulu **group** je nejdříve vytvořena skupina s názvem, který je obsažen ve slovníku **vsftpd_user**. Dále je pomocí modulu **user** vytvořen uživatel pro testování služby. Uživatel patří do skupiny sudo a skupiny, která byla vytvořena v předchozím tasku. Dále je pomocí modulu **apt** stažena a nainstalována služba vsftpd. V následujícím tasku je služba

vsftpd spuštěna. Pomocí argumentu enable bylo povolené automatické spuštění služby vsftpd. V další části byl vytvořen blok, který slouží k zprovoznění SSL/TLS, v případě že je v inventáři ve slovníku **vsftpd_config** definován slovník **ssl**. V rámci zprovoznění SSL/TLS bylo nutné vytvořit task, který vytvořil složku, kde budou uloženy jednotlivé certifikáty. Dále byl pomocí modulu generován privátní klíč. Použitím modulu **openssl_csr** byla vytvořena žádost **CSR** (Certificate Signing Request) za pomocí vytvořeného privátního klíče a definovaného subjektu, pro který bude certifikát vystaven. V neposlední řadě byl vygenerován self-signed certifikát pomocí modulu **openssl_certificate**. Typ certifikátu (např. selfsigned) je nutné specifikovat přímo v argumentu provider. Dále byl vytvořený kořenový adresář ftp pro předem vytvořeného FTP uživatele pomocí modulu **file**. Stejným způsobem byl vytvořen i podadresář test. Další task renderuje Jinja2 šablonu **vsftpd.j2** pro konfiguraci služby vsftpd. V případě, že dojde ke změně konfigurace, tak je pomocí atributu **notify** notifikován **handler** restart vsftpd, který po provedení posledního úkolu restartuje službu vsftpd. Následně je pomocí modulu **copy** vytvořen userlist, ve kterém je uveden seznam uživatelů, kteří mají povolený přístup do FTP(S) serveru.

Na zařízení **Server1** byla provedena konfigurace FTPS serveru pomocí Ansible hry Setup and Configure FTPS server.

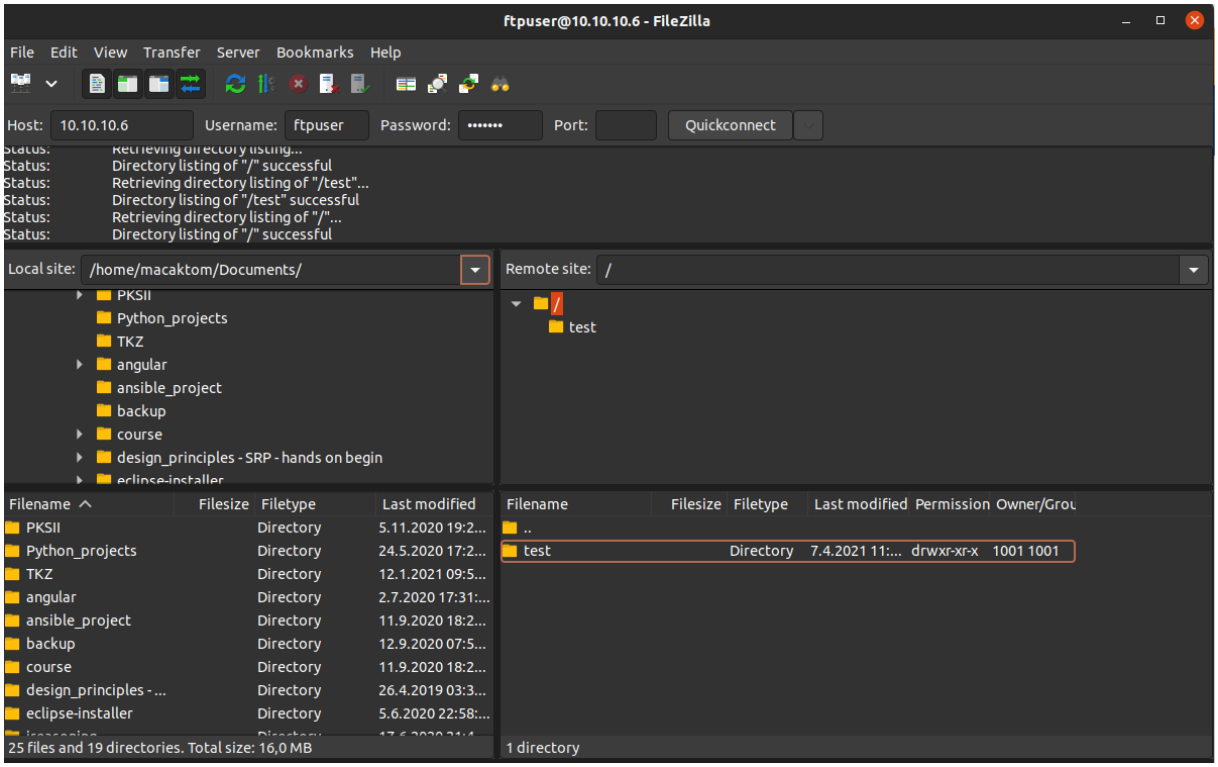
```
//Konfigurace FTPS serveru (služba vsftpd)
$ ansible-playbook linux_configuration.yml --tags vsftpd
```

Po provedení Ansible hry bude vytvořen nový uživatel **ftpuser** s heslem **ftpuser** (dle údajů z inventáře). Dále bude vygenerován self-signed certifikát a nakonfigurován FTPS server. Pro testování FTPS serveru byl použit FTP klient FileZilla. Po přihlášení se zobrazí upozornění na neznámý certifikát FTPS serveru (viz obrázek č. 24).



Obrázek 24: Certifikát FTPS serveru

Po odkliknutí se zobrazí na pravé straně složka **test**, která byla vytvořena pomocí Ansible úkolu **Create test FTP directory** (viz obrázek č. 25).



Obrázek 25: Testování FTPS serveru pomocí FTP klienta FileZilla

4.9.3 Konfigurace TIG stacku

Ansible hra **Setup TIG Stack** je součástí playbooku **linux_configuration.yml** (viz výpis č. 41). Nejdříve jsou implementovány tasky na importování veřejného klíče pro InfluxDB a Grafana repozitáře pomocí modulu **apt_key**. Dále byly přidány jednotlivé repozitáře do souboru /etc/apt/sources.list pomocí modulu **apt_repository**. Dále byla provedena instalace a spuštění služby influxdb. V dalším tasku je provedena instalace balíčku python3-pip. Potom byla provedena instalace Python balíčku influxdb pomocí modulu **pip**, jelikož moduly **community.general.influxdb_user** a **community.general.influxdb_database** jsou závislé na Python balíčku influxdb. V dalším tasku je za pomoci modulu **community.general.influxdb_user** vytvořen nový uživatel pro přístup do databáze, který má administrátorská práva (admin: yes). Potom jsou za využití modulu **community.general.influxdb_database** vytvořeny tři databáze v InfluxDB. První databáze (monitoring_ansible) je používána v Ansible projektu, druhá databáze (monitoring_nornir) je používána pro monitorování síťových zařízení pomocí Nornir projektu a třetí databáze (monitoring_telegraf) je používána Telegraf agentem pro ukládání posbíraných dat. Dále je nainstalován a spuštěn projekt Grafana a Telegraf agent. Potom je renderována Jinja2 šablona **telegraf.j2**, která obsahuje konfiguraci Telegraf agenta. V

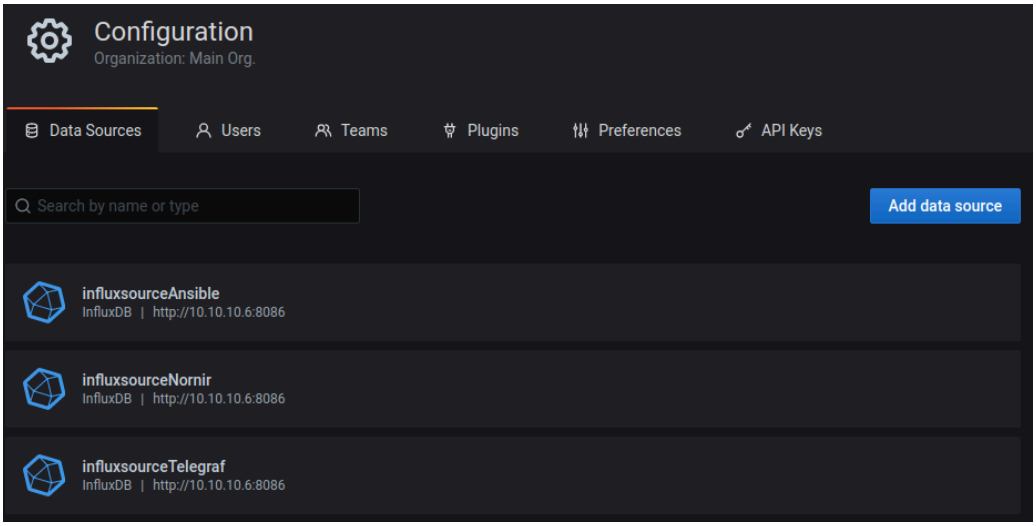
případě změny v konfiguraci je notifikován handler **restart telegraf**, který restartuje službu telegraf po provedení všech Ansible tasků. V posledním tasku jsou pomocí modulu **community.grafana.grafana_datasource** vytvořeny všechny datové zdroje, které byly definovány v inventáři. Jelikož je použito více datových zdrojů, tak je nutno projít datovou strukturu **grafana_data_sources** pomocí direktiva **with_items**. Přístup k prvkům dané iterace je proveden pomocí slovníku item. V inventáři je nutné ve slovníku grafana_data_sources především definovat název datového zdroje, typ datového zdroje (tzn. použitá databáze, např. influxdb), název databáze, InfluxDB URL, jméno a heslo uživatele (pro přístup k databázi).

4.10 Monitorování síťových zařízení a Ubuntu serverů

Pro monitorování síťových zařízení a Ubuntu serverů byly použity nástroje TIG stack a Ansible. Popis implementace Ansible hry pro konfiguraci TIG stacku je uveden v oddílu č. 4.9.3. Popis Jinja2 šablony pro konfiguraci služby telegraf je uveden v oddílu č. 4.9.1. Ansible hra s názvem **Setup TIG Stack** je spuštěna následovně:

```
//Konfigurace TIG stacku
$ ansible-playbook linux_configuration.yml --tags monitoring
```

Po provedení Ansible tasku **Setup TIG Stack** bude na hostu Server1 nakonfigurovaný TIG Stack. Jednotlivé služby jsou provozovány na IP adrese **10.10.10.6** a naslouchají na svých defaultních portech. Zároveň jsou vytvořeny tři databáze (monitoring_ansible, monitoring_nornir, monitoring_telegraf) a datové zdroje (influxsourceAnsible, influxsourceNornir, influxsourceTelegraf). Vytvořené datové zdroje jsou zobrazeny na obrázku č. 26. Dále byl vytvořen jeden uživatel **monitoring** pro jednotný přístup k vytvořeným databázím. Pro přístup k projektu Grafana lze použít URL odkaz: **http://10.10.10.6:3000**.



Obrázek 26: Zobrazení datových zdrojů v projektu Grafana

4.10.1 Sběr a ukládání dat Ubuntu serveru

Pro sběr a ukládání dat se používá služba **telegraf**. Služba je nakonfigurována tak, že na zařízení Server1 je každých 20s proveden sběr dat a jejich následné uložení do databáze monitoring_telegraf. Pro samotný sběr dat jsou použity input pluginy **cpu** a **system**.

4.10.2 Sběr a ukládání dat síťových zařízení

Pro sběr a ukládání dat síťových zařízení byl vytvořen playbook **db_handler.yml** (viz výpis č. 42), který využívá NAPALM knihovnu pro sběr dat a influxdb knihovnu pro uložení dat do InfluxDB databáze monitoring_ansible. Na Cisco zařízeních byl proveden sběr dat o dostupnosti zařízení (přesněji uptime) a vytížení procesoru. Na Juniper JunOS Olive byl proveden pouze sběr dat o dostupnosti zařízení (uptime). Pro pravidelný sběr a zápis dat do databáze vytvořen bash skript **db_playbook_runner.sh** (viz výpis č. 4), který pravidelně spouští playbook **db_handler.yml**.

```
#!/bin/bash

while true
do
   ansible-playbook db_handler.yml
   sleep 1
done
```

Výpis 4: Bash skript db_playbook_runner.sh

Implementace playbooku db_handler.yml:
Playbook **db_handler.yml** (viz výpis č. 42) se skládá z Ansible hry **Get and Write NAPALM data to DB**. Nejdříve je proveden sběr dat z jednotlivých zařízení pomocí NAPALM getterů **environment** (vytížení procesoru) a **facts** (základní údaje o zařízeních). Výsledky jsou uloženy v proměnné **result**. Dále je pomocí příkazu date zobrazen aktuální čas ve formátu Y-m-d H:M:S, který je pomocí argumentů environment a TZ uveden v UTC (Coordinated Universal Time). Jelikož v atributu hosts jsou uvedeny všechny síťové zařízení, tak je nutné tento task delegovat localhostu pomocí argumentu delegate_to (pro uložení identického času pro všechna zařízení). Aktuální čas je uložen v proměnné **date_utc**. Dále je nutné z hodnoty proměnné result vyfiltrovat pouze hodnoty, které chceme zobrazit. Pro Cisco zařízení (a Juniper zařízení vyjma Juniper Olive) byla vyfiltrována hodnota vytížení procesoru. Tento údaj byl uložen do proměnné **cpu_usage**. U všech zařízeních byla podobně vyfiltrována dostupnost zařízení. Výsledná hodnota byla uložena do proměnné **uptime**. Poslední dva tasky **Write HW details into DB** a **Write device facts into DB** využívají modul **community.general.influxdb_write** pro zápis jednotlivých měření do databáze. Zápis je proveden pouze pokud jsou definovány proměnné, které byly vytvořeny v předchozích úkolech (cpu_usage, uptime). V obou úkolech je zápis

časových bodů proveden sériově (host po hostu) nikoliv paralelně. Pro sériový zápis do databáze byl použit atribut **throttle** (od Ansible 2.9), pomocí kterého lze pro jednotlivé tasky definovat, kolik vláken má být použito pro provedení tasku. V modulu community.general.influxdb_write je nutné definovat IP adresu, na které je provozována služba influxdb a název databáze. Pro zápis datových bodu byly v obou úkolech vytvořeny measurement (hw_details, device_facts). V **tags** je uveden klíč **host**. Hodnotou je samotný název zařízení, který se mění v závislosti na zařízení, které provádí daný task. Hodnota časová značky (time) je uložena v proměnné date_utc, která byla definována v tasku **Get current UTC time**. Ve **fields** jsou uvedeny potom samotné měřené hodnoty. V případě tasku **Write HW details into DB** je referencována proměnná cpu_usage. V tasku **Write device facts into DB** je referencována proměnná uptime.

4.10.3 Grafické zobrazení uložených dat Ubuntu serveru

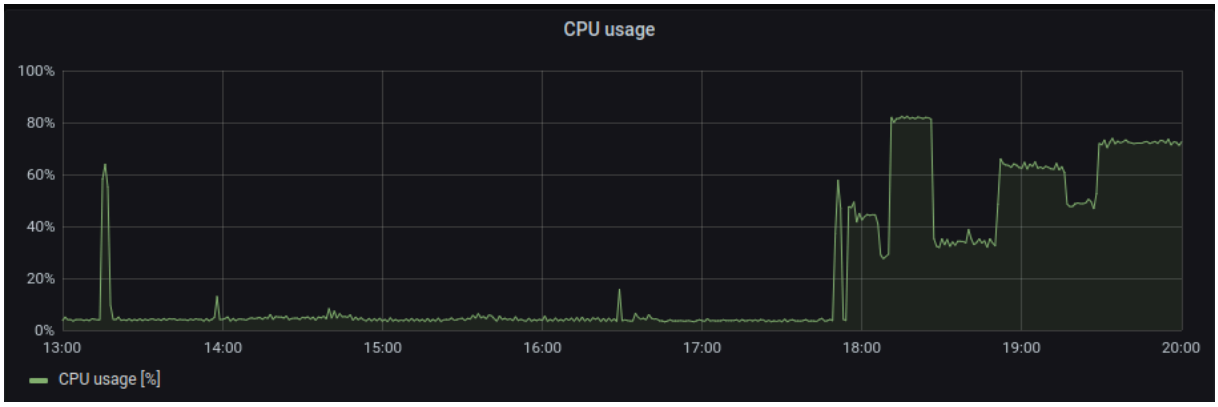
Pro zobrazení uložených dat byla v projektu Grafana vytvořena složka **Telegraf**, která obsahuje dashboard **Server1**. Dashboard pro zařízení Server1 obsahuje dva panely. V projektu Grafana byly tedy pro zařízení Server1 vytvořeny dva grafy: **CPU usage** (viz obrázek č. 27) a **Uptime** (viz obrázek č. 28). Jako datové úložiště byl použit již dříve vytvořený datový zdroj **influx-sourceTelegraf**. Monitorování zařízení bylo provedeno v kratším časovém intervalu (od 13:00 do 20:00), jelikož měření probíhalo na osobním počítači. Pro vytvoření jednotlivých grafů bylo nutné vytvořit následující dotazy v dotazovacím jazyce **InfluxQL**:

```
/* Vytížení procesoru */
SELECT (100 - mean("usage_idle")) FROM "cpu" WHERE ("host" = '10.10.10.6' AND "
    cpu" = 'cpu-total') AND $timeFilter GROUP BY time(1m) fill(null)

/* Uptime */
SELECT last("uptime") FROM "system" WHERE ("host" = '10.10.10.6') AND
    $timeFilter GROUP BY time(1m) fill(null)
```

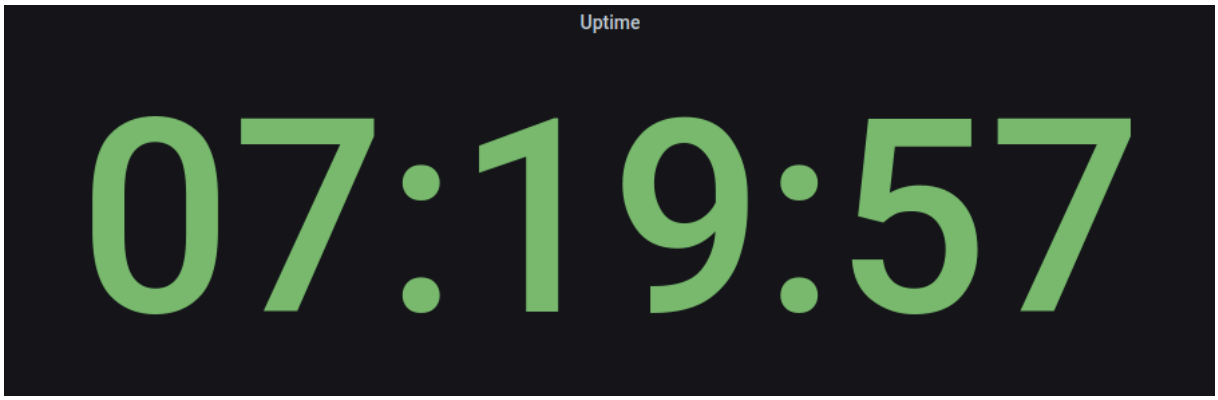
Data jsou do databáze zapsána několikrát za minutu. Pomocí InfluxQL konstrukce **GROUP BY time(1m)** byly seskupeny jednotlivé záznamy podle časového intervalu (v tomto případě 1 minuta). Proměnná **$timeFilter** obsahuje časový interval, který uživatel zvolí pro celý dashboard popřípadě samotný panel.

Pro **výběr hodnoty vytížení procesoru** je nejdříve nutné pomocí tagu **"cpu"='cpu-total'** vyfiltrovat pouze záznamy, které se týkají všech jader procesoru. Z měření **cpu** je potom vybrán field **usage_idle**, na který je aplikována agregační funkce mean(). Vytížení procesoru je potom vypočteno následujícím vztahem: $100 - mean(\text{"}usage\_idle\text{"})$. Field **uptime** je zaznamenán v měření **system**. Jelikož chceme vizualizovat **nejaktuálnější** čas dostupnosti zařízení (uptime), tak je nutné vybrat z databáze poslední přidanou hodnotu dostupnosti pomocí agregační funkce last().

Graf **CPU usage** (viz obrázek č. 27) zobrazuje procentuální vytížení procesoru (osa Y) zařízení Server1 na definovaném časovém intervalu od 13:00 do 20:00 (osa X). Z grafu lze vyčíst, že přibližně v 13:20 narostlo vytížení procesoru (z důvodu provádění příkazu **sudo apt update**). Od 17:45 do 20:00 byla spuštěna služba **stress-ng**, která umožňuje provádět zátěžové testy. Ačkoliv měření probíhalo sedm hodin (od 13:00 do 20:00), tak graf **Uptime** (viz obrázek č. 28) zobrazuje vyšší hodnotu uptime (dostupnosti zařízení), jelikož před sběrem a zápisem dat do InfluxDB databáze bylo nutné nejdříve nakonfigurovat zařízení Server1 (síťová rozhraní, TIG stack).



Obrázek 27: Vytížení procesoru síťového zařízení Server1



Obrázek 28: Uptime síťového zařízení Server1

4.10.4 Grafické zobrazení uložených dat síťových zařízení

Pro zobrazení uložených dat byla v projektu Grafana vytvořena složka Ansible, která obsahuje dashboardy R1, R2, R3 a MLS1. Každý dashboard obsahuje dva panely vyjma dashboardu R2, který obsahuje pouze jeden panel. Pro zařízení R1, R3, MLS1 byly vytvořeny tedy dva grafy: **CPU usage** a **Uptime**. Pro zařízení R2 byl vytvořen pouze graf **Uptime**. Pro zařízení R1 jsou

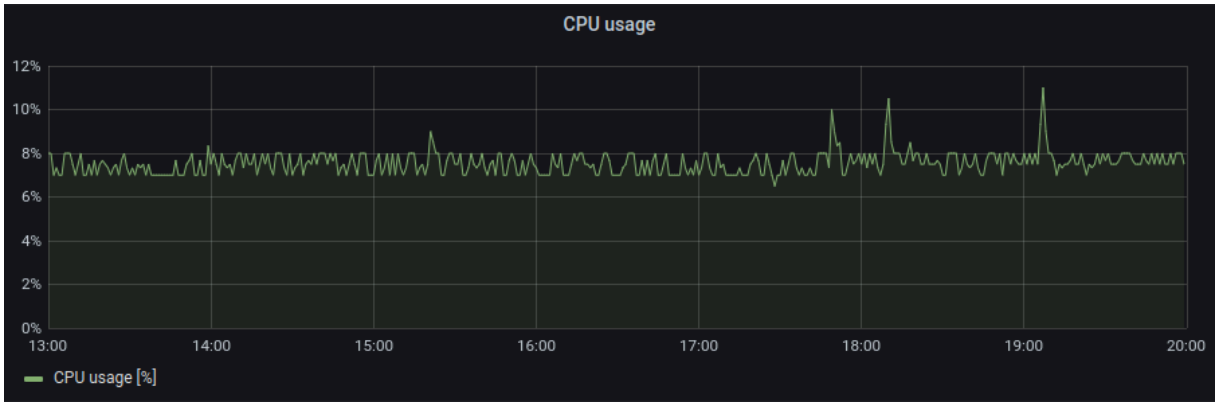
grafy uvedeny na obrázku č. 29 a 30. Pro ostatní zařízení jsou grafy uvedeny v příloze K. Jako datové úložiště byl použit již dříve vytvořený datový zdroj **influxsourceAnsible**. Monitorování zařízení bylo provedeno v kratším časovém intervalu (od 13:00 do 20:00), jelikož měření probíhalo na osobním počítači. Pro vytvoření jednotlivých grafů bylo nutné vytvořit následující dotazy v dotazovacím jazyce **InfluxQL**:

```
/* Vytížení procesoru (zařízení R1) */
SELECT mean("cpu_usage") FROM "hw_details" WHERE ("host" = 'R1') AND
    $timeFilter GROUP BY time(1m) fill(null)

/* Uptime (zařízení R1) */
SELECT last("uptime") FROM "device_facts" WHERE ("host" = 'R1') AND $timeFilter
     GROUP BY time(1m) fill(null)
```

Vytvořené dotazy slouží pro výběr záznamů hosta R1. Pro ostatní zařízení se vytvořené dotazy liší pouze hodnotou tagu **host**, která je používána v InfluxQL klauzuli **WHERE**. Pro zařízení R2 nebyl implementován dotaz, který vybere hodnotu vytížení procesoru. Data jsou do databáze zapsána několikrát za minutu. Pomocí InfluxQL konstrukce **GROUP BY time(1m)** byly seskupeny jednotlivé záznamy podle časového intervalu (v tomto případě 1 minuta). Proměnná **$timeFilter** obsahuje časový interval, který uživatel zvolí pro celý dashboard popřípadě samotný panel.

Pro výběr hodnoty vytížení procesoru je použit field **cpu_usage**, který je součástí měření **hw_details**. Na field cpu_usage byla aplikována agregační funkce mean(). Field **uptime** je součástí měření **device_facts**. Na field uptime je aplikována agregační funkce last(), pomocí které lze vybrat z databáze poslední přidanou hodnotu uptime.

Graf **CPU usage** (viz obrázek č. 29) zobrazuje procentuální vytížení procesoru (osa Y) zařízení R1 na definovaném časovém intervalu od 13:00 do 20:00 (osa X). Ačkoliv měření probíhalo sedm hodin (od 13:00 do 20:00), tak **Uptime** grafy jednotlivých zařízení ukazují vyšší hodnotu dostupnosti zařízení, jelikož síťová zařízení byla zapnutá 30 minut před samotným sběrem a zápisem dat. Důležité je si povšimnout rozdílu při sběru dat o dostupnosti zařízení různých vendorů. Pro Cisco zařízení je vždy zaznamenán uptime s přesností na minuty. U Juniper zařízení (R2) je zaznamenána hodnota s přesností na vteřiny. Způsob vizualizace dat (včetně zaokrouhlení a použitých jednotek) lze však upravit podle potřeby v nástroji Grafana pro každý panel zvlášť.



Obrázek 29: Vytížení procesoru síťového zařízení R1

Obrázek 30: Uptime síťového zařízení R1

5 Návrh automatizovaného systému pro konfiguraci a monitorování síťových zařízení pomocí nástroje Nornir

Vytvořený Nornir projekt **network_automation** je **Python projekt**, který využívá framework Nornir. Při implementaci projektu je kladen důraz na flexibilitu a modularitu řešení. Pro implementaci projektu bylo využito programovací paradigma OOP (Objektově orientované programování). Pro používání projektu **network_automation** je nutné mít nainstalovaný **Python interpreter verze 3.8 a více**. Jednotlivé skripty a inventáře jsou v projektu okomentovány. Dokumentace k projektu je k dispozici ve složce **html**.

5.1 Správa Nornir projektu

Nornir projekt je spravován stejně jako Ansible projekt. Pro správu verzí zdrojového kódu, šablon, zálohovaných konfigurací atd. je použit decentralizovaný verzovací systém Git. Obsah celého projektu je zveřejněn na webové platformě GitLab.[5] V Nornir projektu jsou Python balíčky včetně jejich závislostí spravovány pomocí nástroje Pipenv. Práce s nástrojem Pipenv je popsána v části č. 4.1. Obsah Pipfile souboru je uveden ve výpisu č. 20.

5.2 Struktura vytvořeného Nornir projektu

Nornir projekt je složený z mnoha částí (např. implementované Python moduly, šablony, konfigurace atd.), které dohromady tvoří funkcionalitu řešení, které bylo specifikováno v požadavcích na projekt. Struktura projektu je vyobrazena na obrázku č. 42.

Základní popis struktury projektu:

- **backups** - Složka, ve které jsou uloženy zálohy běžící konfigurace jednotlivých síťových zařízení.
- **export** - Složka, která obsahuje podsložky, pod kterými jsou uloženy exporty obsahu síťové konfigurace a směrovacích tabulek. Dále jsou zde uloženy i jednotlivé reporty (.xlsx soubory).
- **html** - Složka obsahující dokumentaci k „public"metodám a třídám jednotlivých skriptů.
- **initial_setup** - Složka obsahující příkazy, které je nutné provést na jednotlivých zařízeních ještě předtím, než je prováděna automatizace pomocí Norniru (tzn. nastavení doménového jména, konfigurace SSH atd.).
- **inventory** - Složka obsahující inventáře pro Nornir projekt. V **podsložce examples** jsou příklady inventářů, kde jsou popsány jednotlivé parametry (pro každé zařízení zvlášť). Jednotlivé soubory tedy slouží pouze jako dokumentace pro uživatele. V **podsložce host_vars**

[5] `https://gitlab.com/macaktom/network_automation`

lze nalézt soubory, ve kterém jsou definovány parametry konfigurace pro každé zařízení zvlášť. Soubory jsu pojmenovány dle klíčů, které jsou uvedeny v hosts.yml. V **podsložce vault** jsou dva soubory **creds.yml** a **.vault_pass**. V creds.yml jsou uloženy citlivé parametry jednotlivých skupin. Ve .vault_pass je uložené vault heslo, které je použité pro (de)šifrování souboru creds.yml. V **hosts.yml** je uveden seznam cílových síťových zařízení včetně základních parametrů. V **group.yml** je uveden seznam skupin a jejich parametry.

- **modules** - Složka, která obsahuje Python moduly, které lze importovat do hlavního skriptu main.py. V **podsložce tasks** jsou skripty, které slouží ke konfigurování síťových zařízení a serverů. V **podsložce utility** jsou dodatečné skripty, které umožňují sbírat data ze síťových zařízení, provádět export dat atd.
- **templates** - Obsahuje Jinja2 šablony, které slouží pro konfiguraci daných zařízení. Šablony jsou vytvářeny podle vendora (Cisco, Juniper, Debian), případně dle typu zařízení (router, L3_switch, server).
- **config.yml** - Soubor pomocí kterého lze definovat typ inventáře (např. SimpleInventory) včetně nastavení cesty k souborům hosts.yml a group.yml. Dále umožňuje nastavit počet použitých vláken při provádění Nornir úkolu.
- **Pipfile** - Definuje všechny použité knihovny pro development (včetně jejich závislostí).
- **nornir.log** - Log soubor, který zaznamenává události jednotlivých Nornir úkolů. Logování Nornir úkolů je volitelné. V případě, že je logování Nornir tasků povolené, tak není možné pomocí **logging** modulu implementovat vlastní logovací mechanismus.
- **main.py** - Hlavní skript, který se používá pro vytváření objektů ostatních tříd. Dále se používá pro volání metod vytvořených objektů za účelem konfigurace, exportu nebo monitorování stavu daného zařízení.
- **backup_configuration.py** - Skript pro pravidelné zálohování běžící konfigurace všech zařízení (dodatečně je nutné nastavit spuštění tohoto skriptu v nástroji cron).
- **restore_configuration.py** - Skript, který nahrazuje běžící konfiguraci zařízení za dříve zálohovanou konfiguraci, která byla předtím provedena pomocí skriptu backup_configuration.py.
- **db_handler.py** - Skript, který umožňuje pravidelně sbírat data ze síťových zařízení a zapisovat je do InfluxDB databáze.

5.3 Tvorba inventáře a zabezpečení citlivých dat

Pro definování inventáře bylo nejdříve nutné registrovat plugin **SimpleInventory** v souboru **config.yml**. V config.yml bylo dále nutné nastavit cestu k souborům **hosts.yml** a **group.yml**.

Ve slovníku **runners** je nutné aktivovat plugin **threaded**, který povoluje multithreading při spouštění Nornir úkolů. V **hosts.yml** (viz výpis č. 45) je uveden seznam jednotlivých zařízení včetně nejzákladnějších parametrů (type, vendor atd.), které jsou například používány pro filtrování inventáře, tak aby daný Nornir task byl spuštěn pouze pro některé zařízení. Pod klíčem groups jsou dále definovány skupiny, do kterých daná zařízení náleží. Parametry skupin jsou uvedeny v souboru **group.yml** (viz výpis č. 46). V group.yml je nutné definovat klíč platform, který je používán pro inicializaci NAPALM ovladače. Další důležité parametry pro inicializaci ovladače jako username a password jsou definovány v zašifrovaném souboru **creds.yml**, tak aby nebyly citlivé data zobrazena jako prostý text. V connection_options jsou potom definovány dodatečné parametry, které využívají knihovny Netmiko a NAPALM. Mezi tyto atributy patří například global_delay_factor. Patří zde i atribut secret, který bude přidán do connection_options dynamicky. Tak stejně jako atributy username a password, tak i secret je definován v zašifrovaném souboru creds.yml. Pro parsování statického inventáře je nutné v Pythonu inicializovat Nornir objekt pomocí konstruktoru **InitNornir**. V konstruktoru je nutné uvést cestu ke konfiguračnímu souboru config.yml.

```
nr = InitNornir(config_file="config.yml")
```

Proměnná **nr** referencuje Nornir objekt, který obsahuje zparsované inventáře jednotlivých hostů a skupin uvedených v hosts.yml a group.yml. Mimo jiné má implementovanou metodu **run**, která se používá pro paralelní spouštění Nornir tasků. Plugin SimpleInventory není vhodný, pokud konfigurujeme větší množství zařízení. Proto byla implementovaná funkcionalita host_vars, která je dostupná v Ansiblu. Ve složce host_vars jsou tedy vytvořeny soubory, které jsou pojmenovány podle názvů zařízení, která jsou uvedena v hosts.yml. Jednotlivé inventáře jsou uvedeny v příloze L. V každém souboru jsou definovány parametry, které platí pouze pro daného hosta. Pokud tedy konfigurujeme například síťová rozhraní pro IPv4 prostředí, tak je nutné v jednotlivých .yml souborech definovat slovník **interfaces_ipv4** se všemi nutnými parametry. Obsah jednotlivých souborů je definován na základě požadavků, které jsou uvedeny v oddílu č. 3.4. Obsah host_vars souborů je tedy velmi podobný, jako u implementovaného Ansible projektu. V Nornir projektu je však obsah jednotlivých souborů zparsován pouze v případě, že je to nutné pro daný Nornir úkol. Parsování v jednotlivých Nornir úkolech se provádí pomocí Nornir podúkolu, který je volán následovně:

```
data = task.run(task=load_yaml, file=f'inventory/host_vars/{task.host.name}.yml
    ', name="Load host data", severity_level=logging.DEBUG)
```

Pomocí Nornir podúkolu **load_yaml** lze paralelně parsovat YAML inventáře jednotlivých síťových zařízení, které jsou obsaženy v objektu typu Task a jsou tedy referencovány proměnnou **task**. V argumentu file je uvedena cesta k danému YAML souboru v závislosti na názvu zařízení, který je uveden v objektu typu Host (referencován pomocí task.host.name).

Nornir momentálně neumožňuje zabezpečit citlivá data jednotlivých skupin. Tuto funkcionalitu musí tedy programátor implementovat sám. Pro zabezpečení dat byl použit nástroj Ansible Vault. Ve složce **vault** byly vytvořeny dva soubory: .vault_pass a creds.yml. Soubor **creds.yml** (viz výpis č. 5) obsahuje citlivá data (username, password, secret) jednotlivých skupin, která jsou zašifrována pomocí Ansible Vault. Pro šifrování je použito heslo, které se nachází v souboru .vault_pass.

```
---
cisco:
  username: "admin"
  password: "automationDP"
  secret: "cisco"
juniper:
  username: "juniper"
  password: "Juniper"
linux:
  username: "gns3"
  password: "gns3"
  secret: "gns3"
```

Výpis 5: Dešifrovaný soubor creds.yml

Pro zparsování statického inventáře a citlivých dat byla implementována ve spustitelných skriptech (main.py, backup_configuration, restore_configuration, db_handler) funkce **setup_inventory** (viz výpis č. 55). Nejdříve je pomocí konstruktoru inicializován Nornir objekt, který obsahuje data ze zparsovaného statického inventáře hosts.yml a group.yml. Potom je inicializován objekt typu **CredentialHandler**. Po zavolání konstruktoru dojde k dešifrování souboru creds.yml pomocí importované knihovny ansible a souboru .vault_pass. Dešifrovaný obsah je uložen v instanční proměnné self._credentials. Dále je ve funkci setup_inventory zavolána metoda **insert_creds**. Do metody insert_creds je vložena reference na Nornir objekt. Pomocí implementované metody insert_creds lze do zparsovaného inventáře dynamicky vložit dešifrovaný obsah souboru creds.yml. Po provedených změnách v Nornir objektu vrací funkce setup_inventory referenci na upravený Nornir objekt. Pro ilustraci provedených změn lze porovnat v projektu soubory **group.yml** a **group_show_creds.yml**.

Nornir objekt obsahuje zparsovaný inventář všech zařízení. Pokud chceme spustit Nornir úkol pouze na některých zařízeních, tak je nutné vyfiltrovat zařízení pomocí metody **filter** a **F** objektu. F objekt umožňuje filtrovat zařízení podle názvu, skupiny nebo jakéhokoliv parametru, který je specifikován v inventáři. Filtrování lze provést následovně:

```
def main() -> None:
  #Parsování inventáře
```

```
nornir_obj = setup_inventory()

#Filtrování podle klíče dev_type (pouze směrovače)
routers = nornir_obj.filter(F(dev_type="router"))

#Filtrování podle názvu skupiny (skupina juniper)
juniper_devices = nornir_obj.filter(F(groups__contains="juniper"))

#Filtrování podle názvu zařízení (zařízení MLS1 a R3)
mls1_r3 = nornir_obj.filter(F(name__contains="MLS1") | F(name__contains="R3"
    ))
```

Další filtrované objekty jsou uvedeny ve výpisu č. 55.

5.4 Konfigurace síťových zařízení

Pro konfiguraci síťových zařízení v Norniru je nejdříve nutné mít nadefinovaný inventář. Dále je nutné vytvořit Jinja2 šablony pro jednotlivé konfigurace. V neposlední řadě je nutné implementovat Nornir úkoly (funkce), které umožňují zařízení konfigurovat.

5.4.1 Tvorba Jinja2 šablon

Proces vytváření Jinja2 šablon pro Nornir projekt se velmi podobá tvorby Jinja2 šablon pro Ansible projekt (viz oddíl č. 4.5.1). Šablony, které byly vytvořeny v Ansible projektu, tak byly překopírovány do Nornir projektu. Jediná nutná úprava je pro přístup k jednotlivým parametrům v inventáři. V případě Norniru jsou všechny parametry hosta referencovány pomocí proměnné **host** (objekt typu **Host**). Pokud jsou tedy vytvářeny šablony pro konfiguraci OSPFv3 (viz výpisy č. 52 a 53), tak pro přístup k slovníku **ospfv3_config** je nutné použít proměnnou **host** (např. host.ospfv3_config). Kromě přístupu k parametrům hosta jsou šablony zcela identické.

5.4.2 Implementace Nornir tasků

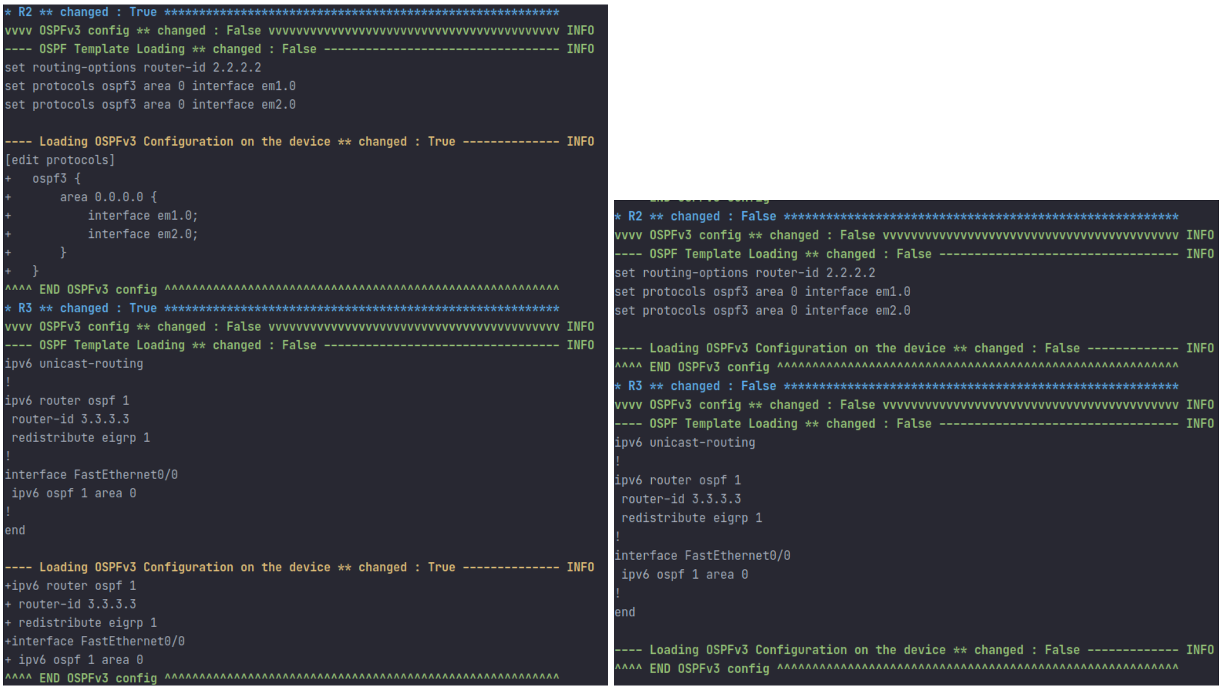
Pro konfiguraci síťových zařízení byly implementovány následující třídy: InterfacesConfiguration, EIGRPConfiguration, OSPFConfiguration, PacketFilterConfiguration, StaticRoutingConfiguration a NATConfiguration. Implementace jednotlivých tříd je velmi podobná. Většina těchto tříd obsahuje dva Nornir úkoly. První Nornir úkol slouží pro konfigurování pro IPv4 prostředí a pomocí druhého úkolu lze provést konfiguraci pro IPv6 prostředí. Příkladem je například implementace hromadného konfigurování OSPF. Konfigurace OSPF je implementována ve třídě **OSPFConfiguration** (viz výpis č. 56). Třída obsahuje dva Nornir úkoly: **configure_ospf** a **configure_ospfv3**. Nornir úkol je implementován tak, že nejdříve se načtou data jednotlivých

hostů z YAML inventářů. V případě Nornír úkolu configure_ospfv3 se po zparsování inventáře zkontroluje, jestli v načtených datech je uveden klíč **ospfv3_config**. Pokud je uveden, tak načtený obsah slovníku ospfv3_config je uložen do objektu typu Host (referencován pomocí task.host). Pomocí podúkolu **template_file** je provedeno renderování Jinja2 šablon. Obsah renderované šablony je uložen do Host objektu pod klíčem **ipv6_ospf**. Renderovaná šablona je potom použita v podúkolu **napalm_configure**, který slouží ke konfiguraci síťových zařízení pomocí knihovny NAPALM. Pro konfiguraci je použita operace merge (replace=False). Do argumentu configuration je uvedena reference k obsahu renderované šablony. Pomocí argumentu **dry_run** lze spustit Nornir task v testovacím režimu. V testovacím režimu obdrží uživatel pouze rozdíly mezi novou a běžící konfigurací. Objekt třídy **OSPFConfiguration** je potom inicializován v spustitelném skriptu **main.py** (viz výpis č. 55). Mimo jiné byla vytvořena wrapper funkce **configure_network_devices**, která vykoná požadovaný Nornir task na daném Nornir objektu a vypíše výsledky jednotlivých podúkolů do konzole pomocí Nornir funkce **print_result**. Příklad konfigurace OSPFv3 může vypadat následovně:

```
def main() -> None:
    nornir_obj = setup_inventory()
    routers = nornir_obj.filter(F(dev_type="router"))
    ospf_config = OSPFConfiguration()

    #Konfigurace OSPFv3 - testovací režim
    configure_network_devices(routers, ospf_config.configure_ospfv3, "OSPFv3
        config", dry_run=True)

    #Konfigurace OSPFv3
    configure_network_devices(routers, ospf_config.configure_ospfv3, "OSPFv3
        config", dry_run=False)
```

5.4.3 Testování konfigurace síťových zařízení

V rámci testování (viz oddíl č. 3.4) bylo nutné ve skriptu main.py (viz výpis č. 55) v metodě **main** zparsovat inventář pomocí funkce setup_inventory. Dále bylo nutné vyfiltrovat Nornir objekt, tak aby například konfigurace paketových filtrů byla provedena pouze pro zařízení MLS1. Dále byly inicializovány Configuration objekty (InterfacesConfiguration, OSPFConfiguration atd.). V neposlední řadě byla pro jednotlivé konfigurační Nornir úkoly volána funkce **configure_network_devices**. Po provedení konfigurace je do konzole vepsán obsah renderované šablony a rozdíly mezi novou konfigurací a běžící konfigurací (viz obrázek č. 31).



(a) Po prvním provedení Nornir tasku (b) Po druhém provedení Nornir tasku

Obrázek 31: Výpis rozdílů v konfiguracích po vykonání Nornir tasku configure_ospfv3 (R2, R3)

5.5 Sběr dat ze síťových zařízení

Pro sběr dat ze síťových zařízení a jejich následný výpis do konzole byla implementována třída **NetworkUtilityViewer**. Třída **NetworkUtilityViewer** má implementované Nornir úkoly, které slouží například pro výpis směrovacích tabulek, OSPF sousedů, IP adres jednotlivých síťových rozhraní, základních údajů o síťových zařízení atd. Jednotlivé Nornir úkoly byly implementovány za pomoci podúkolů **napalm_get** nebo **netmiko_send_command**. Příkladem prvního typu implementace je Nornir task **show_vlans**, který se používá pro zobrazení konfigurovaných VLAN. Sběr dat je proveden pomocí tasku napalm_get a NAPALM getteru get_vlans. Pomocí příznaku **json_out** může uživatel definovat, jestli strukturované data chce vepsat do konzole jako formátovaný JSON string nebo se pro výpis do konzole použije pomocná Nornir funkce **print_result**. V případě že pro danou situaci neexistuje potřebný NAPALM getter, tak je pro sběr dat použit podúkol **netmiko_send_command**. Příkladem je implementovaný Nornir task **show_ospf_neighbors**, který slouží pro zobrazení OSPFv2 případně OSPFv3 sousedů. Podúkol **netmiko_send_command** využívá knihovnu Netmiko k paralelnímu zasílání příkazů jednotlivým zařízením. V Nornir tasku **show_ospf_neighbors** bylo tedy nutné definovat jednotlivé příkazy podle vendora zařízení. Pokud je argument **ipv6** roven True, tak je na Cisco zařízeních proveden příkaz **show ipv6 ospf neighbor** (vyjma Cisco IOSvL2) a na Juniper zařízeních příkaz **show ospf3 neighbor**. V opačném případě je na Cisco zařízeních vykonán příkaz **show ip ospf neighbor** a na Juniper zařízeních **show ospf neighbor**. Po pro-

vedení příkazu jsou obdržená nestrukturovaná data vypsána do konzole pomocí Nornir funkce **print_result**.

Pro testování sběru a následného výpisu dat do konzole bylo nutné v **main.py** inicializovat objekt typu **NetworkUtilityViewer**. Spuštění jednotlivých tasků je provedeno pomocí **run** metody Nornir objektu

```
def main() -> None:
    nornir_obj = setup_inventory()
    viewer = NetworkUtilityViewer()
    l3_switches = nornir_obj.filter(F(dev_type="L3_switch"))
    l3_switches.run(task=viewer.show_vlans, json_out=False)
```

Task **show_vlans** byl odzkoušen na zařízení MLS1. Po provedení tasku **show_vlans** jsou vypsány nakonfigurované VLANy pro zařízení MLS1 (viz obrázek č. 32). Některé další tasky pro sběr a výpis dat včetně možnostech jejich spuštění jsou uvedeny ve spustitelném skriptu main.py (viz výpis č. 55).

```
**** Device MLS1: ************************************************************
---- MLS1: Show configured VLANs ** changed : False ---------------------------- INFO
{ 'vlans': { '1': { 'interfaces': [ 'GigabitEthernet1/1',
                                    'GigabitEthernet1/2',
                                    'GigabitEthernet1/3',
                                    'GigabitEthernet2/0',
                                    'GigabitEthernet2/1',
                                    'GigabitEthernet2/2',
                                    'GigabitEthernet2/3',
                                    'GigabitEthernet3/0',
                                    'GigabitEthernet3/1',
                                    'GigabitEthernet3/2'],
                     'name': 'default'},
             '1002': {'interfaces': [], 'name': 'fddi-default'},
             '1003': {'interfaces': [], 'name': 'token-ring-default'},
             '1004': {'interfaces': [], 'name': 'fddinet-default'},
             '1005': {'interfaces': [], 'name': 'trnet-default'},
             '30': {'interfaces': ['GigabitEthernet0/2'], 'name': 'VLAN30'},
             '40': {'interfaces': ['GigabitEthernet0/3'], 'name': 'VLAN40'},
             '50': {'interfaces': ['GigabitEthernet1/0'], 'name': 'VLAN50'}}}
```

Obrázek 32: Výstup po provedení Nornir tasku show_vlans - MLS1

5.6 Exportování dat a tvorba reportů

Pro exportování dat a tvorbu reportů byla implementována třída **NetworkInfoExporter**. Exportovat lze například běžící konfigurace, směrovací tabulky nebo pravidla pro filtrování paketů. Export dat do souborů s příponou .conf nebo .txt je vždy proveden paralelně. Dále lze vytvářet Excel reporty. První typ Excel reportu sumarizuje základní informace o síťových zařízení. Na druhou stranu druhý typ Excel reportu poskytuje základní statistiku o přijatých a vysílaných paketech na síťových rozhraní. Při vytváření reportů jsou data nejdříve sbírána paralelně, vzápětí jsou však data jednotlivých zařízení agregována. Tato data jsou potom zapsána do excelovského sešitu (ve formát .xlsx).

5.6.1 Implementace Nornir úkolů pro exportování dat

Pro exportování dat byly implementovány Nornir tasky **export_device_configuration**, **export_ipv4_routes**, **export_ipv6_routes** a **export_packet_filter_info**. Nornir tasky jsou implementovány tak, že je nejdříve paralelně proveden sběr potřebných dat pomocí knihovny NAPALM případně Netmiko. V některých Nornir úkolech je dále provedeno parsování posbíraných dat pomocí implementovaných metod třídy **NetworkInfoParser**. Pro zápis dat do .txt nebo .conf souboru byla implementovaná třída **FileExporter**, která obsahuje metodu **export_to_file**.

5.6.2 Implementace metod pro tvorbu Excel reportů

Pro tvorbu Excel reportů byly implementovány dvě metody: **export_device_facts** a **export_interfaces_packet_counters**. V obou případech se jedná o metody, které paralelně vykonávají pouze Nornir úkol pro sběr dat. Všechny ostatní operace jsou vykonány sériově. Nejdříve bylo nutné definovat proměnnou sorted_headers_export, ve které bude uložena hlavička tabulky (nadpisy). Dle seřazené hlavičky budou seřazena i ostatní data daného zařízení. Nejdříve jsou paralelně posbírany data z jednotlivých zařízení. Posbírána data jsou agregována do objektu typu **AggregatedResult**, který obsahuje posbírané data všech zařízení. Tento objekt je referencován pomocí proměnné all_results_aggregation. Pomocí implementovaných metod třídy **NetworkInfoParser** je provedeno parsování AggregatedResult objektu. O vytvoření XLSX souboru a zápis dat se stará inicializovaný objekt typu **ExcelExporter**, který využívá knihovnu **openpyxl**. Při tvorbě Excel reportu pomocí metody **export_device_facts** jsou sumarizovány základní informace síťových zařízení v jednom excelovském listu. V případě tvorby statistik vysílaných a přijatých paketů pomocí **export_interfaces_packet_counters** je pro každé zařízení vytvořen samostatný excelovský list. Název listu odpovídá názvu síťového zařízení (tzn. R1, R2 atd.).

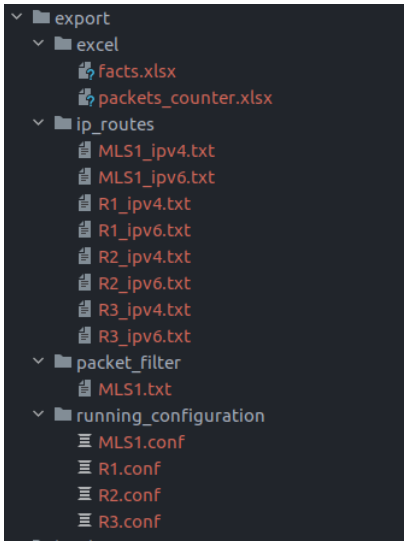
5.6.3 Testování Nornir tasků a metod

Pro testování exportování dat a tvorbů reportů bylo nutné v metodě main (viz výpis č. 6) zparsovat jednotlivé inventáře, vyfiltrovat Nornir objekt, inicializovat objekt typu NetworkInfoExporter a zavolat příslušné tasky/metody, které jsou implementované v třídě NetworkInfoExporter. Pro exportování dat jsou zavolány jednotlivé Nornir tasky pomocí metody run. V případě tvorby reportů je reference na objekt typu Nornir vložena do argumentu implementovaných metod.

```
def main() -> None:
    nornir_obj = setup_inventory()
    l3_devices = nornir_obj.filter(F(dev_type="router") | F(dev_type="L3_switch"
        ))
    exporter = NetworkInfoExporter(NetworkInfoCollector())

    #Export dat
    l3_devices.run(task=exporter.export_device_configuration)
    l3_devices.run(task=exporter.export_packet_filter_info)
    l3_devices.run(task=exporter.export_ipv4_routes)
    l3_devices.run(task=exporter.export_ipv6_routes)
     #Tvorba Excel reportů
    exporter.export_device_facts(l3_devices)
    exporter.export_interfaces_packet_counters(l3_devices)
```

Výpis 6: Metoda main pro testování exportu dat a tvorbu reportů

Po provedení následujících tasků a metod vznikne složka export, ve které jsou vytvořeny podsložky, které obsahují soubory s exportovanými daty (viz obrázek č. 33):



Obrázek 33: Struktura složky export

Po provedení metod **export_device_facts** a **export_interfaces_packet_counters** byly v podsložce excel vytvořeny dva excelovské sešity: **facts.xlsx** (viz obrázek č. 34) a **packets_counter.xlsx** (viz obrázek č. 35).

hostname	FQDN	vendor	model	serial_number	os_version	uptime	connection
R1	R1.automation.local	Cisco	7206VXR	4279256517	Version 15.2(4)S5	5:36:00	OK
R2	R2.automation.local	Juniper	OLIVE	-	12.1R1.9	5:42:22	OK
R3	R3.automation.local	Cisco	7206VXR	4279256517	Version 15.2(4)S5	5:34:00	OK
MLS1	MLS1.automation.local	Cisco	IOSv	9LD1YQMD0KM	Version 15.2(4.0.55)E	5:42:00	OK

Obrázek 34: Excel report facts.xlsx

interface	rx_broadcast	rx_discards	rx_errors	rx_multicast	rx_octets	rx_unicast	tx_discards	tx_errors	tx_octets	tx_unicast
em0	-1	0	0	-1	514565	-1	0	0	922454	-1
em1	-1	0	0	-1	31734	-1	0	0	33276	-1
em2	-1	0	0	-1	31710	-1	0	0	32912	-1
mtun	-1	-1	-1	-1	0	-1	-1	-1	0	-1

Obrázek 35: Excelovský list R2 souboru packets_counter.xlsx

5.7 Zálohování, obnovení a mazání konfigurace

V rámci Nornir projektu bylo implementováno zálohování, obnovení i mazání konfigurace. Pro zálohování a obnovení konfigurace byly vytvořeny spustitelné skripty **backup_configuration.py** a **restore_configuration.py**. Pro mazání části konfigurace byly implementovány Nornir tasky ve skriptu **delete_configuration.py**. Inicializace objektu typu **DeleteConfiguration** je provedena ve spustitelném skriptu **main.py**.

5.7.1 Zálohování běžící konfigurace

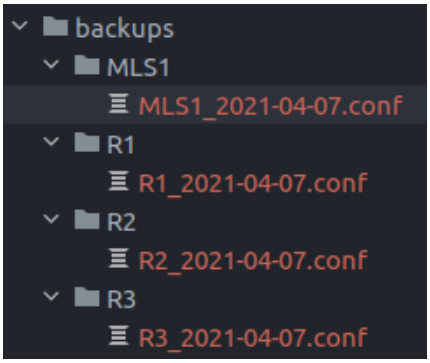
Pro zálohování běžící konfigurace byl vytvořen spustitelný skript **backup_configuration.py**. Ve skriptu backup_configuration.py je implementována třída **BackupConfiguration**. V průběhu inicializace objektu typu BackupConfiguration je do instanční proměnné **self._date** uloženo aktuální datum. Třída BackupConfiguration má implementovaný Nornir task **backup_device_running_configuration**, který umožňuje paralelně provádět zálohu běžící konfigurace jednotlivých zařízení. Pomocí NAPALM getteru **get_config** je ze síťových zařízení získán obsah běžící konfigurace. Dále jsou definované relativní cesty k výsledným souborům vzhledem ke kořenovém adresáři projektu. Pro definované cesty budou vytvořeny všechny nadřazené složky, v případě že tyto adresáře již neexistují. Paralelní zápis dat do souborů je proveden pomocí Nornir podúkolu **write_file**.

Pro testování skriptu je nutné inicializovat objekt typu BackupConfiguration, zparsovat inventáře, vyfiltrovat Nornir objekt dle potřeby a v argumentu metody **run** deklarovat Nornir task backup_device_running_configuration.

```
if __name__ == '__main__':
```

```
backup_configuration = BackupConfiguration()
nr = backup_configuration.setup_inventory()
all_devices = nr.filter(F(dev_type="router") | F(dev_type="L3_switch") | F(
    dev_type="switch"))
all_devices.run(backup_configuration.backup_device_running_configuration,
    name="Backup running configuration")
```

Po spustění skriptu **backup_configuration.py** vznikne složka backups a dodatečné podsložky, kde lze nalézt jednotlivé zálohy (viz obrázek č. 36):



Obrázek 36: Struktura složky backups

5.7.2 Mazání konfigurace

Pro mazání části konfigurace byl implementován Nornir task **delete_configuration** ve třídě **DeleteConfiguration**. Ve složce **templates** byly vytvořeny Jinja2 šablony **delete_configuration.j2**, ve kterém jsou specifikovány jednotlivé příkazy, které jsou použity pro smázání jednotlivých částí konfigurace (např. síťová rozhraní, OSPF, EIGRP atd.). V inventáři musí uživatel ve slovníku **delete_config** specifikovat, které části konfigurace budou později smazány. Pro Cisco zařízení jsou tyto šablony vytvořeny pomocí imperativního přístupu, jelikož šablony obsahují příkazy, které začínájí se slovem no. Nornir task **delete_configuration** nejdříve zparsuje inventáře jednotlivých zařízení pomocí podúkolu **load_yaml** a do objektu typu Host dynamicky uloží pod klíčem **delete_config** hodnotu slovníku delete_config. Dále je provedeno renderování šablon delete_configuration.j2. Obsah renderované šablony je uložen do objektu typu Host pod klíčem **conf_delete**. Pomocí podúkolu napalm_configure jsou aplikovány na zařízení jednotlivé příkazy, které jsou obsaženy v renderované šabloně. Implementovaný task lze spustit v testovacím režimu.

Pokud chceme otestovat mazání konfigurace, tak je nutné nadefinovat v inventáři ve slovníku **delete_config** jednotlivé části konfigurace, které chceme smazat. Dále je nutné v metodě main zparsovat inventář, vyfiltrovat Nornir objekt, inicializovat objekt třídy **DeleteConfiguration**

a zavolat Nornir task delete_configuration. Pokud chceme tedy u všech zařízení odstranit části konfigurace, které byly provedeny v oddílu č. 5.4.3, tak je nutné jednotlivé části konfigurace pro daná síťová zařízení definovat v delete_config (viz příloha L).

```
def main() -> None:
    nornir_obj = setup_inventory()
    l3_devices = nornir_obj.filter(F(dev_type="router") | F(dev_type="L3_switch"
        ))
    delete_config = DeleteConfiguration()
    configure_network_devices(l3_devices, delete_config.delete_configuration, "
        Delete Configuration", dry_run=False)
```

Výpis 7: Implementace metody main pro testování mazání konfigurace

5.7.3 Obnovení zálohované konfigurace

Pro obnovení zálohované konfigurace byl vytvořen spustitelný skript **restore_configuration.py**. Obnovení konfigurace je implementováno, tak aby mohl uživatel nahradit běžící konfiguraci za zálohovanou konfiguraci dle data, které je uvedeno v inventáři jednotlivých hostů. Nornir task **restore_running_configuration**, implementovaný ve třídě **RestoreConfiguration**, umožňuje paralelně nahradit stávající konfiguraci jednotlivých zařízení za již zálohovanou konfigurace. Nejdříve je zparsován inventář jednotlivých hostů. Do proměnné **date** je uložené datum, které je definované v inventáři v **running_config_date**. Dále je definována cesta k jednotlivým zálohám. Obsah jednotlivých konfigurací je uložen do objektu typu **Host**. Pomocí podúkolu **napalm_configure** je provedeno samotné nahrazení konfigurace. Pro nahrazení konfigurace je nutné argumentu **replace** přiřadit hodnotu True.

Pro nahrazení běžící konfigurace za zálohovanou konfiguraci je nutné mít nejdříve vytvořenou nějakou zálohu konfigurace (viz oddíl č. 5.7.1). V předchozím oddílu bylo provedeno smazání jednotlivých částí konfigurace. Ve skriptu **restore_configuration.py** je nutné inicializovat objekt typu **RestoreConfiguration**, zparsovat inventáře, vyfiltrovat Nornir objekt a metodě **run** přidat do argumentu Nornir task restore_running_configuration (viz výpis č. 8).

```
if __name__ == '__main__':
    restore_conf = RestoreConfiguration()
    nr = restore_conf.setup_inventory()
    all_devices = nr.filter(F(dev_type="router") | F(dev_type="L3_switch") | F(
        dev_type="switch"))
    res = all_devices.run(restore_conf.restore_running_configuration, name="
        Restore backed up configuration",dry_run=False)
    print_result(res)
```

Výpis 8: Testování obnovení zálohované konfigurace

V inventáři jednotlivých zařízení je nutné definovat slovník **restore_config**. Jelikož záloha byla provedena 7. dubna 2021, tak je nutné v inventářích definovat slovník restore_config následovně:

```
restore_config:
  running_config_date: 2021-04-07
```

Po spuštění skriptu **restore_configuration.py** je nahrazena běžící konfigurace na jednotlivých zařízeních za konfiguraci, které byla zálohována 7. dubna 2021.

5.8 Konfigurace Ubuntu serverů

Pro konfiguraci zařízení s operačním systémem Ubuntu 18.04 byla implementována třída **LinuxConfiguration**. V třídě LinuxConfiguration jsou implementovány dva Nornir tasky: **send_commands** a **configure_vsftpd**. Nornir task **send_commands** umožňuje pomocí knihovny Netmiko vzdáleně vykonat příkazy na linuxových serverech. Příkazy jsou definovány v inventáři jako list. List příkazů je uložen v proměnné **commands**. Pro provedení příkazů je použit podúkol **netmiko_send_command**. Výsledek provedeného příkazu je vypsán do konzole. Nornir task byl testován na zařízení Server1. Nejdříve bylo nutné v inventáři zařízení Server1 (viz výpis č. 51) definovat list příkazů, které se mají provést (např. příkazy pwd, ls). V **main.py** je nutné zparsovat inventář, vyfiltrovat Nornir objekt, inicializovat objekt typu LinuxConfiguration a přiřadit do argumentu metody **run** task send_commands.

```
def main() -> None:
    nornir_obj = setup_inventory()
    linux_config = LinuxConfiguration()
    ubuntu_servers = nornir_obj.filter(F(dev_type="ubuntu_server") | F(
        groups__contains="linux"))
    ubuntu_servers.run(task=linux_config.send_commands, enable=True)
```

Po provedení Nornir tasku **send_commands** byly vypsány do konzole výsledky příkazů **pwd** a **ls** pro zařízení Server1 (viz obrázek č. 37).

```
**** Device Server1: ************************************************************
/home/gns3
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
```

Obrázek 37: Výpis výsledků příkazů pwd a ls po provedení tasku send_commands

5.8.1 Konfigurace FTPS serveru

Pro konfiguraci FTPS serveru byl implementován Nornir task **configure_vsftpd**. Implementovaný task zparsuje inventář jednotlivých hostů. Obsah slovníku **vsftpd_config** je pro každé zařízení uloženo do objektu typu Host. List příkazů uvedených ve **vsftpd_config.commands** je uložen do proměnné **commands**. Dále je provedeno renderování šablony **vsftpd.j2** (viz výpis č. 54). Výsledná konfigurace **vsftpd.conf** je exportována do podsložky configuration, která se nachází ve složce export. Dále jsou pomocí podúkolu **netmiko_send_command** provedeny jednotlivé příkazy, které jsou obsaženy v proměnné commands.

Na zařízení Server1 byl nejdříve manuálně vytvořen uživatel ftpuser. Dále byla manuálně vytvořena složka /etc/**certs** a self-signed certifikát. Self-signed certifikát byl vytvořen pomocí knihovny OpenSSL následovně:

```
$ sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/certs/
    vsftpd.pem -out /etc/certs/vsftpd.pem
```

V inventáři hosta Server1 (viz výpis č. 51) je definován slovník **vsftpd_config**, který obsahuje konfiguraci pro TLS (důležité při renderování šablony vsftpd.j2) a jednotlivé příkazy, které se na zařízení provedou pomocí knihovny Netmiko. Pro konfiguraci FTPS serveru je nutné v metodě **main** zparsovat inventář, inicializovat objekt typu LinuxConfiguration, vyfiltrovat Nornir objekt a metodě run přiřadit task configure_vsftpd.

```
def main() -> None:
    nornir_obj = setup_inventory()
    linux_config = LinuxConfiguration()
    ubuntu_servers = nornir_obj.filter(F(dev_type="ubuntu_server") | F(
        groups__contains="linux"))
    ubuntu_servers.run(task=linux_config.configure_vsftpd, enable=True)
```

Task **configure_vsftpd** nejprve zprovozní službu **vsftpd**. Dále vytvoří kořenový adresář **ftp** (/home/ftpuser/ftp) a testovací složku **test** (/home/ftpuser/ftp/test). Potom dojde k smazání defaultního konfiguračního souboru **vsftpd.conf**. Pokud je v inventáři definován příkaz **scp**, tak dojde k přenosu vytvořeného konfiguračního souboru **vsftpd.conf** z PC_mgmt na Server1 včetně následného přesunu souboru do /etc/ (/etc/vsftpd.conf). Mimo jiné se provede i změna vlastníka konfiguračního souboru na **root**. Dále jsou provedeny příkazy, které jsou definovány v inventáři hosta (vytvoření userlistu a restartování služby vsftpd). Stav konfigurace a služby vsftpd není kontrolován (imperativní přístup ke konfiguraci).

5.9 Monitorování síťových zařízení a Ubuntu serverů

Pro monitorování síťových zařízení a Ubuntu serverů byly využity nástroj **TIG stack** a **Nornir**. TIG stack je nutné konfigurovat manuálně nebo využít implementované Ansible hry **Setup TIG**

Stack (viz oddíl č. 4.10).

5.9.1 Sběr a ukládání dat Ubuntu serverů

Pro sběr a ukládání dat se používá služba **telegraf**. Nezávisí tedy na použitém automatizačním nástroji (Ansible, Nornir), jelikož má vše na starosti Telegraf agent. Služba je nakonfigurována tak, že pro zařízení Server1 je každých 20s proveden sběr dat a jejich následné uložení do databáze **monitoring_telegraf**. Služba **telegraf** byla konfigurována pomocí Ansible hry **Setup TIG Stack**.

5.9.2 Sběr a ukládání dat síťových zařízení

Pro sběr a ukládání dat sítových zařízení do InfluxDB databáze byl vytvořen spustitelný skript **db_handler.py** (viz výpis č. 57). Ve skriptu je implementovaná třída **DBHandler**, pomocí které lze sbírat a zapisovat data síťových zařízení do InfluxDB databáze (konkrétně monitoring_nornir). Mimo jiné lze pomocí implementovaných metod zobrazit nebo smazat záznamy z vytvořených databází. Třída **DBHandler** má implementovanou metodu **write_monitored_data**, která v nekonečné smyčce sbírá data pomocí Nornir úkolu **napalm_get** a NAPALM getterů **get_environment**, **get_facts**. Ostatní operace jsou prováděny sériově. Výsledný objekt typu **AggregatedResult** je procházen for cyklem. Data každého hosta jsou potom zparsována pomocí metody **get_monitored_fields_values**, která vrací slovník měřených hodnot (fields) konkrétního hosta. Pomocí metody **write_to_db** je proveden samotný zápis datových bodů do InfluxDB databáze (v tomto případě monitoring_nornir).

Na Cisco zařízeních byl proveden sběr dat o dostupnosti zařízení (respektivě uptime) a vytížení procesoru. Na Juniper JunOS Olive byl proveden pouze sběr dat o dostupnosti zařízení (uptime).

5.9.3 Grafické zobrazení uložených dat Ubuntu serveru

Grafické zobrazení uložených dat Ubuntu serveru bylo popsáno v části č. 4.10.3.

5.9.4 Grafické zobrazení uložených dat síťových zařízení

Pro zobrazení uložených dat byla v projektu Grafana vytvořena složka Nornir, která obsahuje dashboardy R1, R2, R3 a MLS1. Jako datové úložiště byl použit již dříve vytvořený datový zdroj **influxsourceNornir**. Další postup je ekvivalentní postupu, který byl použit v Ansible řešení. Popis měření, InfluxQL dotazů a grafů byl již proveden v části č. 4.10.4. Vytvořené grafy jsou uvedeny v příloze O.

6 Zhodnocení a porovnání jednotlivých řešení

6.1 Zhodnocení Ansible projektu

Tento oddíl se zabývá celkovým zhodnocením vytvořeného systému, který je implementován pomocí automatizačního nástroje Ansible a monitorovacího nástroje TIG stack.

6.1.1 Automatizovaná konfigurace síťových zařízení a sběr dat

Pomocí vytvořeného projektu lze hromadně konfigurovat emulovaná zařízení Cisco c7200, Juniper JunOS Olive a Cisco IOSvL2. Pro sběr strukturovaných dat z jednotlivých zařízení byl použit modul **napalm_get_facts**. Je nutné však zdůraznit, že některé NAPALM gettery např. **get_environment, get_interfaces_counters** nejsou zcela podporované emulovaným zařízením Juniper JunOS Olive. Situace by však mohla být jiná, pokud by se testovalo například na Juniper zařízeních řady vSRX. Jelikož **napalm_get_facts** implementuje pouze několik filtrů, tak bylo nutné například pro zobrazení směrovacích tabulek využít modul **napalm_cli**, který umožňuje provést definované CLI příkazy. Obdržená nestrukturovaná data musela být zparsována. V rámci projektu bylo otestováno nahrazení celé běžící konfigurace za zálohovanou konfiguraci. Problémy nastaly při nahrazování konfigurace emulovaného zařízení Cisco IOSvL2, u kterého je nutné si dávat pozor především při nahrazení konfigurace síťových rozhraní (nahrazení **no negotiation auto** za **negotiation auto** a **no switchport** za např. **switchport access vlan 10**). Příčinou tohoto problému může být buď použitý obraz emulovaného zařízení Cisco IOSvL2 nebo samotná implementace modulu **napalm_install_config**. Co se týče operace sloučení konfigurací (merge operace), tak u operačního systému Cisco IOS nefunguje v některých případech deklarativní přístup ke konfigurací síťových zařízení. Týká se to zejména při nastavení síťových rozhraní (např. **switchport** a **no switchport**, **shutdown** a **no shutdown**). V takovém případě pokud chceme nastavit switch port, tak příkaz **switchport** musí být součástí Jinja2 šablony, i když se samotný příkaz nezobrazí v běžící konfiguraci.

6.1.2 Automatizovaná konfigurace Ubuntu serveru

V rámci Ansible projektu byla odzkoušena i konfigurace serveru s operačním systémem Ubuntu 18.04. Uživatel může na serveru hromadně zprovoznit FTPS server s certifikátem, který je podepsaný sám sebou. Dále lze pomocí vytvořené Ansible hry automaticky nakonfigurovat celý monitorovací nástroj TIG (Telegraf, InfluxDB, Grafana) stack včetně automatického nastavení datových zdrojů v nástroji Grafana.

6.1.3 Export dat a vytváření reportů

Vytvořený Ansible projekt lze použít i pro exportování dat a vytváření různých reportů. Data jsou získána pomocí modulů **napalm_get_facts** a **napalm_cli**. Uživatel například může ex-

portovat obsah směrovací tabulky do .txt souboru. Dále lze exportovat i běžící konfigurace do .conf souboru. Jednotlivé exporty jsou prováděny paralelně. V rámci Ansible ekosystému momentálně není k dispozici modul na vytváření Excel reportů v XLSX formátu. Pro vytváření reportů je použit šablonovací systém Jinja2. Po renderování šablony je vytvořen report ve formátu HTML. Příkladem takového reportu je sumarizace informací o každém síťovém zařízení.

6.1.4 Monitorování testovaných zařízení

Pro monitorování zařízení byl použit **TIG stack**. Pro zápis dat síťových zařízení do InfluxDB databáze byl vytvořen Ansible scénář, který je pravidelně proveden dle implementovaného **bash skriptu**. Pokud jsou při zápisu dat referencovány proměnné pomocí Jinja2 konstrukce, tak jsou dané data (včetně časového razítka) reprezentována jako textový řetězec. V takovém případě je možné využít rozšíření **jinja2_native**, které povolí Jinja2 nativní datové typy, tak aby datové typy referencovaných proměnných nebyly vždy přetypovány na string. Při použití tohoto řešení je však nutné počítat s tím, že pokud budete chtít data pouze vypsat do příkazového řádku, tak je nutné toto rozšíření zakázat, jelikož yaml callback plugin neumí pracovat s Jinja2 nativními datovými typy. Pomocí nástroje Grafana byla úspěšně provedena vizualizace získaných dat. Pomocí vytvořeného řešení lze monitorovat uptime a vytížení procesoru jednotlivých zařízení vyjma zařízení R2.

6.1.5 Shrnutí

Vytvořený Ansible projekt splňuje požadavky, které byly kladeny na systém v části č. 3. Pomocí vytvořeného řešení lze hromadně konfigurovat emulovaná zařízení Cisco c7200, Juniper JunOS Olive a Cisco IOSvL2. S menšími úpravami lze použít tento systém i pro konfiguraci ostatních zařízení, které využívají stejný operační systém (Cisco IOS, Junos OS). Vytvořené řešení trpí nedostatky, které však především vychází z omezených možností automatizace správy síťových zařízení, které používají CLI a použitých obrazů emulovaných zařízení (nepodporované a Juniper JunOS Olive neoficiální). Ansible projekt lze použít i pro konfiguraci vybraných služeb serveru s operačním systémem Ubuntu 18.04. Pomocí vytvořeného systému lze testovaná zařízení i monitorovat (uptime a vytížení procesoru). Nedostatky monitorovací části lze pozorovat především v nutnosti využívat rozšíření jinja2_native při zápisu datových bodů do InfluxDB, jelikož v takovém případě nebude při provedení jiných Ansible scenářů korektně fungovat výpis do konzole. Mimo jiné Juniper Olive nepodporuje některé filtry modulu **napalm_get_facts**.

6.2 Zhodnocení Nornir projektu

Tento oddíl se zabývá celkovým zhodnocením vytvořeného systému, který je implementován pomocí automatizačního nástroje Nornir a monitorovacího nástroje TIG stack. Pro Nornir projekt byla vygenerována dokumentace z komentářů tříd, instančních proměnných a metod.

6.2.1 Automatizovaná konfigurace síťových zařízení a sběr dat

Pomocí vytvořeného projektu lze hromadně konfigurovat emulovaná zařízení Cisco c7200, Juniper JunOS Olive a Cisco IOSvL2. Pro hromadnou správu síťových zařízení byly implementovány vlastní Nornir úkoly, které mají definované Nornir podúkoly, které umožňují získavat data z inventáře, renderovat šablony a sloučit konfiguraci s běžící konfigurací síťového zařízení. Pro sloučení nové konfigurace s běžící konfigurací byl využit task **napalm_configure**. Co se týče operace sloučení konfigurací, tak v případě operačního systému Cisco IOS nefunguje v některých případech deklarativní přístup ke konfigurací síťových zařízení. Týká se to zejména při nastavení síťových rozhraní (např. **switchport** a **no switchport, shutdown a no shutdown**). V takovém případě pokud chceme nastavit switch port, tak příkaz **switchport** musí být součástí Jinja2 šablony, i když se samotný příkaz nezobrazí v běžící konfiguraci. Projekt podporuje i vytváření záloh síťové konfigure včetně jejich obnovy pomocí argumentu replace, který je definován v tasku **napalm_configure**. Problémy nastaly při nahrazování konfigurace emulovaného zařízení Cisco IOSvL2, u kterého je nutné si dávat pozor při nahrazení konfigurace síťových rozhraní (nahrazení **no negotiation auto** za **negotiation auto** a **no switchport** za např. **switchport access vlan 10**). Příčinou tohoto problému může být použitý obraz emulovaného zařízení Cisco IOSvL2 nebo v implementaci napalm_configure tasku. Při sběru dat byly zejména využívány **NAPALM gettery**, které lze definovat v argumentu **napalm_get** tasku. Pro obraz Juniper JunOS Olive nejsou podporovány některé NAPALM gettery (např. **environment**). Pokud nebyla nalezena potřebná NAPALM getter metoda, tak bylo nutné využít task **netmiko_send_command**, který však oproti **napalm_get** vrací nestrukturovaná data. Data lze vypsat do konzole pomocnou Nornir metodou **print_result** nebo vytvořenou funkcí, která jednotlivé Nornir výsledky (Result objekty) zformátuje a vypíše jako JSON string.

6.2.2 Automatizovaná konfigurace Ubuntu serveru

Pomocí implementovaného systému lze konfigurovat službu vsftpd na Ubuntu serveru. Dále lze vzdáleně provádět na serveru definované příkazy. Momentálně framework Nornir nenabizí pluginy pro deklarativní konfigurování Ubuntu serverů. Pomocí knihovny Netmiko lze vzdáleně zasílat příkazy, které se mají provést na Ubuntu serveru. Stav konfigurace není kontrolován (imperativní přístup ke konfiguraci). Nornir framework není tedy v této fázi vývoje vhodný pro konfigurování serverů.

6.2.3 Export dat a vytváření reportů

Systém umožňuje exportovat data do souborů s příponou .txt a .conf. Paralelně lze například exportovat obsah směrovací tabulky daných síťových zařízení. Pro samotný zápis souborů byla implementována třída **FileExporter**, která obsahuje důležitou metodu **export_to_file**. Pro vytváření Excel reportů byla implementována třída **ExcelExporter**, která obsahuje potřebné metody pro zápis do souboru a vytvoření souboru. Vytvořená třída využívá knihovnu **openpyxl**,

která slouží pro správu .xlsx souborů. Uživatel může například vytvořit report, který obsahuje základní informace o jednotlivých síťových zařízení.

6.2.4 Monitorování testovaných zařízení

Monitorování síťových zařízení a serverů je provedeno pomocí frameworku Nornir a TIG stacku. Pro pravidelný sběr a následný zápis dat do InfluxDB databáze byla vytvořena třída DBHandler, která je implementována v samostatném a spustitelném Python skriptu **db_handler.py**. Pro Ubuntu servery lze využít Telegraf agenta pro sběr a zápis dat. Pro zprovoznění TIG stacku je nutné TIG stack manuálně nakonfigurovat nebo využít Ansible hry, která je součástí Ansible playbooku **linux_configuration.yml**. Vizualizace uložených dat je provedena pomocí vizualizačního nástroje Grafana. Pomocí vytvořeného řešení lze monitorovat uptime a vytížení procesoru jednotlivých zařízení (vyjma zařízení R2).

6.2.5 Shrnutí

Vytvořený Nornir projekt splňuje požadavky, které byly kladeny na systém v části č. 3. Implementovaný Nornir projekt lze společně s nástrojem TIG stack využít pro hromadnou konfiguraci síťových zařízení, sběr dat a monitorování síťových zařízení případně serverů s operačním systémem Ubuntu 18.04. Pomocí vytvořeného řešení lze hromadně konfigurovat emulovaná zařízení Cisco c7200, Juniper JunOS Olive a Cisco IOSvL2. S menšími úpravami lze použít tento systém i pro konfiguraci ostatních zařízení, které využívají stejný operační systém (Cisco IOS, Junos OS). Nornir framework momentálně nemá implementované pluginy, které umožňují provádět konfiguraci serveru deklarativním přístupem. Nornir tedy není vhodný pro hromadnou konfiguraci serverů. Ostatní nedostatky projektu vychází zejména z použitých obrazů emulovaných síťových zařízení (pouze pro testování, nepodporované) a omezeních, která se vztahují ke konfigurování síťových zařízení, které využívají pouze CLI. Pomocí vytvořeného systému lze testovaná zařízení i monitorovat (uptime a vytížení procesoru).

6.3 Porovnání obou řešení

V této části je provedeno porovnání obou řešení (Ansible a Nornir projekt). Zmíněny jsou zejména rozdíly v implementační části obou projektů. Důležitým aspektem pro výběr správného automatizačního nástroje je rychlost provedení jednotlivých tasků. Rychlost provedení tasků nebude však v diplomové práci zohledněna, jelikož se liší zejména při konfiguraci velkého množství síťových zařízení (>=100).

6.3.1 Tvorba inventáře a zabezpečení citlivých dat

V obou řešeních jsou statické inventáře reprezentovány v datovém formátu YAML. V **Ansible projektu** byl použit i datový formát INI (soubor hosts), který je použit pouze pro specifikování jednotlivých síťových zařízení. Pro parametry jednotlivých hostů byla použita složka

host_vars, která obsahuje YAML soubory s parametry jednotlivých hostů. Názvy jednotlivých souborů jsou ekvivalentní k aliasům hostů, které jsou používané v souboru hosts. Pro specifikování parametrů skupin byla použita složka **group_vars**, která obsahuje složky, jejichž názvy jsou ekvivalentní k aliasům skupin, které jsou používané v souboru hosts. V těchto složkách je soubor vars, který obsahuje všechny parametry dané skupiny. V **Nornir projektu** byl použit defaultní plugin **SimpleInventory**, pomocí kterého lze definovat inventář v souborech hosts.yml a group.yml. V **hosts.yml** jsou definovány spravované síťová zařízení včetně základních parametrů těchto zařízení (např. hostname, groups, vendor atd.). V **group.yml** jsou definovány jednotlivé skupiny a parametry, které náleží těmto skupinám. Nornir momentálně nepodporuje funkcionalitu při použití host_vars a group_vars adresářů pro definování parametrů síťových zařízení a jejich skupin. Tato funkcionalita byla v rámci vytvořeného řešení implementována pro **host_vars**. Obsah souborů, které jsou uloženy v **host_vars**, jsou dynamicky přidány do již zparsovaného inventáře (Nornir objektu).

Pro zabezpečení citlivých dat v inventářích lze v Ansiblu využít **Ansible Vault** pomocí které lze zašifrovat obsah celého souboru nebo jen některých citlivých dat. V Ansible projektu byl pro každou skupinu vytvořen vault soubor, který obsahuje všechny citlivé údaje dané skupiny (hesla, secrets). Definované vault proměnné jsou referencovány v samotných vars souborech. Každy vault soubor je zašifrován za pomoci vault hesla, které je uvedeno v souboru **.vault_pass**. Cesta k souboru **.vault_pass** je konfigurována v konfiguračním souboru **ansible.cfg**. Na rozdíl od Ansiblu, Nornir framework nenabízí žádnou možnost jak zabezpečit citlivé údaje jednotlivých skupin. Citlivá data skupin byla tedy zašifrována pomocí **Ansible Vault**. Vault heslo je uloženo ve **.vault_pass** souboru. Dále byla implementována třída **CredentialHandler**, pomocí které lze dešifrovat citlivé údaje (za pomoci **.vault_pass** souboru) a dynamicky dešifrované údaje vložit do již zparsovaného inventáře (Nornir objektu).

6.3.2 Sběr dat ze síťových zařízení

V obou projektech byly pro sběr dat využity zejména NAPALM gettery. V některých případech (např. při výpisu obsahu směrovací tabulky) bylo nutné v obou projektech definovat příkazy, které byly provedeny na síťových zařízeních. V Ansible projektu byl použit modul **napalm_cli**, který využívá knihovnu NAPALM. V Nornir projektu byl použit task **netmiko_send_command**, který využívá knihovnu Netmiko. Zmíněný task a modul vrací nestrukturovaná data, která jsou zparsována pomocí vytvořených regulárních výrazů a následně vypsána do konzole. Pro výpis dat do konzole je v Ansible projektu použit yaml callback plugin. U Nornir projektu si lze pomocí argumentu **json_out** vybrat, jestli uživatel chce data vypsat pomocí pomocné Nornir funkce **print_result** nebo jako JSON string. U některých metod není výpis pomocí JSON stringu podporován kvůli špatné čitelnosti.

Co lze vypsat do konzole:

1. Běžící konfiguraci
2. NTP (Network Time Protocol) a SNMP informace
3. Vytvořené uživatele
4. Stav připojení síťových zařízení
5. Vytížení HW komponent síťových zařízení (s výjimkou Juniper Olive)
6. Obsah směrovacích tabulek (IPv4, IPv6) vyjma L2 switchů
7. Vytvořené paketové filtry včetně jednotlivých pravidel (IPv4, IPv6)
8. Konfigurované VLAN - pouze switche
9. Základní informace o síťových rozhraní
10. Konfigurované IP adresy na síťových rozhraní (IPv4, IPv6)
11. Statistika vysílaných a přijatých paketů na jednotlivých síťových rozhraní
12. Zobrazení OSPFv2 nebo OSPFv3 sousedů (OSPFv3 není podporován na emulovaném zařízení Cisco IOSvL2)

6.3.3 Tvorba šablon

U obou projektů byl pro vytváření šablon použit šablonovací systém Jinja2. Šablony se u obou projektů liší pouze ve způsobu referencování renderovaných proměnných. U Ansiblu lze v Jinja2 šablonách přímo referencovat proměnné, které jsou definovány v inventářích (viz obrázek č. 38b). U Norniru je inventář hosta referencován v Jinja2 šablonách pomocí tzv. host objektu. Host objekt obsahuje všechny proměnné konkrétního hosta, který je definován v inventáři (viz obrázek č. 38a).

```
{% if host.static_routing_config is defined %}
{% for net in host.static_routing_config.networks %}
ip route {{ net.dest_ip }} {{ net.dest_mask }} {{ net.next_hop }}
{% endfor %}
{% if host.static_routing_config.default is defined %}
ip route 0.0.0.0 0.0.0.0 {{ host.static_routing_config.default.next_hop }}
{% endif %}
!
end
{% endif %}
```

(a) Nornir projekt

```
{% if static_routing_config is defined %}
{% for net in static_routing_config.networks %}
ip route {{ net.dest_ip }} {{ net.dest_mask }} {{ net.next_hop }}
{% endfor %}
{% if static_routing_config.default is defined %}
ip route 0.0.0.0 0.0.0.0 {{ static_routing_config.default.next_hop }}
{% endif %}
!
end
{% endif %}
```

(b) Ansible projekt

Obrázek 38: Ukázka šablony konfigurace statického směrování v IPv4 pro směrovače firmy Cisco

6.3.4 Konfigurace síťových zařízení

Oba projekty využívají pro hromadnou konfiguraci síťových zařízení šablonovací systém Jinja2 a moduly/tasky, které využívají NAPALM knihovnu. Jediný podstatný rozdíl nastává pouze při renderování Jinja2 šablon. U Ansible projektu jsou renderované Jinja2 šablony, které jsou následně uloženy do složky **staging**, ze které jsou potom „nahrány"jednotlivé konfigurační soubory do síťových zařízení. Po dokončení konfigurace jsou jednotlivé konfigurační soubory smazány ze složky **staging**. Naproti tomu u Nornir projektu lze obsah renderované šablony dynamicky vložit (přiřadit) do již zparsovaného inventáře (Nornir objektu).

Co lze pomocí projektů hromadně konfigurovat:

1. Síťová rozhraní (L2 i L3 rozhraní) - konfigurace v IPv4 i IPv6 prostředí.
2. Směrovací protokoly - EIGRP, OSPF, statický routing - konfigurace v IPv4 i IPv6 prostředí (EIGRP a OSPFv3 není podporován zařízením Juniper Olive)
3. Paketové filtry - Vytváření filtrů a definování pravidel, konfigurace v IPv4 i IPv6 prostředí
4. Source NAT Overload - pouze L3 zařízení firmy Cisco

Mimo základní konfiguraci lze běžící konfiguraci zálohovat a obnovit (zcela nahradit za běžící konfiguraci). V neposlední řadě lze uživatelem definované části konfigurace mazat. Mazání jednotlivých částí konfigurace je u obou projektů řešena imperativním přístupem (tzn. provedením jednotlivých příkazů krok za krokem bez kontroly stavu konfigurace). Při hromadné konfiguraci lze nastavit, jestli má být konfigurace provedena v testovacím režimu. Povolením testovacího režimu (**dry_run**) nedojde k sloučení/nahrazení konfigurace, avšak uživatel obdrží konečné změny, které by byly danou operací provedeny. V případě, že není daná část konfigurace u síťového zařízení podporována (např. OSPFv3 u Juniper Olive) nebo korektně sloučena/nahrazena, tak je o tom uživatel informován v konzoli nebo log souboru.

6.3.5 Konfigurace Ubuntu serverů

Ansible je zejména používán pro konfiguraci linuxových serverů. Pomocí vytvořeného Ansible projektu lze hromadně konfigurovat například FTPS server (služba vsftpd) a všechny komponenty TIG stacku včetně nastavení datových zdrojů u Grafany. Ansible playbook **linux_configuration.yml** byl implementován tak, aby veškeré Ansible hry byly idempotentní.

Pro konfiguraci Ubuntu serverů lze u Norniru využít pouze netmiko plugin, který umožňuje vykonávat příkazy na vzdálených zařízeních. Pro odzkoušení konfigurace Ubuntu serverů byla implementovaná třída **LinuxConfiguration**, která obsahuje metody pomocí kterých užívatel může vykonávat příkazy na vzdáleném serveru nebo konfigurovat FTPS server bez možnosti

vytvoření testovacího FTP uživatele a self-signed certifikátu. Pro konfigurování serverů jsou použité příkazy, které jsou definovány v inventáři. Stav konfigurace není na serveru kontrolován.

6.3.6 Export dat a vytváření reportů

V obou projektech je možnost paralelně exportovat obsah směrovacích tabulek a definovaná pravidla vytvořených paketových filtrů do souboru s příponou .txt. Dále je možné paralelně exportovat obsah běžící konfigurace do souboru s příponou .conf. V Ansible projektu je provedena implementace exportu pomocí modulu **copy**. Na rozdíl od Ansible projektu je v Nornir projektu implementována třída **FileExporter**, která má implementované metody pro export dat do souboru s příponou .txt nebo .conf. Třída se nachází v souboru **text_file_exporter.py**. Pomocí obou projektů vytvářet reporty, které obsahují základní údaje o síťových zařízení. Dále lze vytvářet i statistiky, které se týkají přijatých a vysílaných paketů na jednotlivých síťových rozhraní daných zařízení. V Nornir projektu jsou reporty vytvářeny ve formě excelového souboru s příponou .xlsx. Pro implementaci byla použita knihovna **openpyxl**, která je využívána v třídě **ExcelExporter** (v souboru **excel_exporter.py**). Součástí Ansible ekosystému momentálně není modul, který umožňuje spravovat .xlsx soubory. Pro implementaci reportů byly použity Jinja2 šablony, podle kterých byly vytvořeny výsledné **HTML (Hypertext Markup Language) dokumenty**.

6.3.7 Monitorování testovaných zařízení

V obou řešeních je využit **TIG stack**, který lze však nakonfigurovat pouze pomocí vytvořené Ansible hry (nebo manuálně). Data z Ubuntu serveru jsou posbírána pomocí **Telegraf agenta**. Sběr dat je proveden pomocí definovaných **NAPALM getterů**. V Nornir projektu byla pro správu InfluxDB (zápis/mazání datových bodů, zobrazení stavu databáze) použita **knihovna influxdb**. V Ansible projektu je nutné stáhnout a přidat cestu ke stažené kolekci **community.general**. Součástí kolekce je modul **community.general.influxdb_write**, který se používá pro zápis datových bodů do InfluxDB databáze. V konfiguračním souboru **ansible.cfg** je však nutné povolit direktivu **jinja2_native**, jelikož je pro referencování proměnných použit Jinja2 výraz, který automaticky vždy na konci přetypuje daný datový typ na string. Povolením direktivy lze do databáze zapisovat data, která jsou například datového typu integer nebo float. V případě Ansible projektu lze playbook **db_handler.yml** spouštět opakovaně pomocí vytvořeného bash skriptu **db_playbook_runner.sh**. Pro vizualizaci dat je použit projekt Grafana. Na jednotlivých zařízeních je monitorován uptime a vytížení procesoru.

6.3.8 Metody spuštění skriptů/playbooků

Vytvořené Nornir řešení je komplexním Python projektem, který obsahuje vytvořené moduly a tři spustitelné skripty. Hlavní skript tvoří soubor **main.py**, ve kterém uživatel musí v metodě **main** (viz výpis č. 9) inicializovat Nornir objekt, který obsahuje zparsovaný inventář a

možnost spouštět tasky. Dále je nutné vyfiltrovat pouze potřebná zařízení, na kterých budou tasky provedeny. Pro filtrování lze využít metodu **filter**, která je implementována pro **Nornir objekt** a speciální **F** objekt, který se používá pro filtrování síťových zařízení dle klíčů vytvořených v statickém inventáři. V neposlední řadě je nutné zavolat patřičný task pro vyfiltrované zařízení. Využívá se tzv. **run** metody. **Metoda run obsahuje argument task, kterému je přiřazena reference metody, která má být paralelně volána**. V případě konfigurování síťových zařízení byla vytvořena wrapper funkce **configure_network_devices**, která „obaluje"**run** metodu ještě o výpis **Result** objektu do konzole. Hlavní skript **main.py** lze spustit přes příkazový řadek (viz výpis č. 3). Nejlépe je však při upravování projektu a spouštění skriptů použít IDE jako např. PyCharm.

```
def main() -> None:
    # Inicializace Nornir objektu
    nornir_obj = setup_inventory()

    # filtrování síťových zařízení
    routers = nornir_obj.filter(F(dev_type="router"))

    # Inicializace objektu pro konfiguraci síťových rozhraní
    interfaces_configuration = InterfacesConfiguration()

    # Inicializace objektu pro sběr dat
    viewer = NetworkUtilityViewer()

    # Zobrazit údaje o směrovačích
    routers.run(task=viewer.show_device_facts, json_out=False)

    # Zobrazit OSPFv2 sousedy směrovačů
    routers.run(task=viewer.show_ospf_neighbors, ipv6=False)

    # Konfigurace síťových rozhraní (IPv4) všech routerů, wrapper funkce
    configure_network_devices(routers, interfaces_configuration.
        configure_ipv4_interfaces, "IPv4 interfaces config", dry_run=False)
```

Výpis 9: Postup pro práci se skriptem main.py

Vytvořený Ansible projekt se skládá hned z několika playbooků. V rámci jednoho playbooku je implementováno několik Ansible her, které jsou označeny pomocí **tag atributu**. Na základě **tag atributu** lze při spuštění playbooku vyfiltrovat Ansible hry, které se mají provést (viz výpis č. 10). Pokud **tag atribut** není specifikován, tak se provedou všechny Ansible hry, které jsou definovány v playbooku. Například playbook **ospf_configuration.yml** obsahuje dvě Ansible hry

(**OSPFv2 configuration**, **OSPFv3 configuration**). Každá z těchto Ansible her má přiřazený svůj tag, pomocí kterého lze spustit pouze konkrétní Ansible hru.

```
//Konfigurace OSPFv2 a OSPFv3
$ ansible-playbook ospf_configuration.yml

//Konfigurace OSPFv2 a OSPFv3 - testovací režim
$ ansible-playbook ospf_configuration.yml --check

// Konfigurace OSPFv2
$ ansible-playbook ospf_configuration.yml --tags ipv4

// Konfigurace OSPFv3
$ ansible-playbook ospf_configuration.yml --tags ipv6
```

Výpis 10: Možnosti spouštění playbooku

6.3.9 Náročnost implementace

Ansible projekt byl výrazně jednodušší na implementování. Největším problémem byla implementace pro vytváření reportů. Oproti Norniru není Ansible tak flexibilní (např. absence modulu pro vytváření .xlsx souborů). Oproti tomu Nornir využívá programovací jazyk Python. Při práci s Nornirem je nutné pochopit celý jeho ekosystém (pluginy, tasky, subtasky, Result objekty atd.). Využití celého ekosystému Pythonu nabízí však mnohem větší flexibilitu při implementaci oproti nástroji Ansible. V Pythonu lze mnohem lépe diagnostikovat chyby v kódu.

6.3.10 Dokumentace k projektům

Ansible projekt je zejména zdokumentován pomocí názvů jednotlivých Ansible her a tasků. V některých místech projektu jsou využity i komentáře (např. popis inventářů). V případě Nornir projektu byly používány komentáře a dokumentační řetězce. Z dokumentačních řětezců byla vygenerována dokumentace ve formě statických HTML stránek pomocí knihovny pdocs. Vygenerovaná dokumentace nezahrnuje spustitelné skripty a implementované „private"metody. Vygenerované soubory jsou dostupné ve složce html.

6.4 Možnosti rozšíření projektů

Vytvořené projekty lze dále rozšiřovat, na základě měnících se požadavků uživatele. Velmi zajímavým vylepšením jednotlivých projektů by bylo poskytnutí automatizovaného ověřování síťové konfigurace pomocí Nornir tasku napalm_validate a modulu napalm_validate. Uživatel by však musel vytvořit YAML soubor, ve kterém jsou deklarovány **NAPALM getter metody a konfigurační data dle specifické struktury**. Pomocí úkolu/modulu napalm_validate je provedeno

ověření konfigurace v YAML souboru proti výsledkům, které byly obdrženy pomocí definovaných NAPALM getter metod. Celý proces by šel automatizovat tak, že by konfigurační data byla posbírána z YAML inventáře. Uživatel by tak nemusel vytvářet manuálně nové YAML soubory pouze pro validování.

V rámci monitorování síťových zařízení lze například řešení rozšířit o grafickou vizualizaci příchozích/odchozích/ztracených paketů. Momentálně lze tyto data pouze exportovat do .xlsx souboru. Monitorování Ubuntu serverů lze rozšířit například o monitorování vytížení jednotlivých jader procesoru nebo o monitorování velikosti určitých adresářů (např. /var).

Závěr

Hlavním cílem diplomové práce bylo navrhnout a ověřit automatizovaný systém pro konfiguraci a monitorování síťových zařízení a serverů. Implementace systému byla provedena ve dvou variantách: **Ansible + TIG stack** a **Nornir + TIG stack**. Implementované projekty jsou spravovány nástroji **Git** a **Pipenv**. Ansible projekt **network_automation_ansible** je veřejně publikován na platformě **GitLab** (`https://gitlab.com/macaktom/network_automation_ansible`). Veřejně byl publikován i Nornir projekt **network_automation** (`https://gitlab.com/macaktom/network_automation`).

První kapitola byla věnována problematice automatizace síťových konfigurací a správy síťových zařízení. V diplomové práci byly popsány nástroje, technologie a protokoly, které jsou spjaté s automatizací síťové konfigurace a hromadnou správou síťových zařízení a serverů. Důraz byl kladen na automatizační nástroje Nornir a Ansible, které byly dále použity v praktické části diplomové práce. Druhá kapitola se věnuje monitorováním síťových zařízení. Nejprve byl popsán protokol SNMP včetně jeho nedostatků, které jsou adresovány principem Streaming telemetry. Důraz byl v této kapitole kladen na popis moderního přístupu k monitorování síťových zařízení a serverů téměř v reálném čase. V rámci práce byly popsány možnosti sběru dat časových řad, jejich uložení a grafické zobrazení. Podrobněji byly popsány nástroje, které se používají v rámci TIG stacku, který je dále využíván v praktické části diplomové práce.

V praktické části bylo cílem navrhnout, implementovat a ověřit automatizovaný systém pro konfiguraci a monitorování síťových zařízení, které využívají CLI. V rámci praktické části byly provedeny dvě implementace daného systému a to pomocí nástrojů **Ansible + TIG stack** a **Nornir + TIG stack**. Nejdříve byla provedena analýza automatizovaného systému, ve které byly popsány požadavky na daný systém včetně výběrů použitých technologií a nástrojů. Dále byla popsána testovací síťová topologie a použitá emulovaná síťová zařízení v GNS3. Obě řešení založené na nástroji Ansible a Nornir lze použít pro sběr dat, konfiguraci, zálohování/obnovení/-mazání konfigurace, exportování dat do .txt a .conf souborů nebo vytváření souhrnných reportů (.html nebo .xlsx). Konfigurace byly vytvářeny pomocí Jinja2 šablon. V rámci monitorování zařízení byl nejdříve proveden sběr dat ze síťových zařízení (pomocí knihovny NAPALM) a serveru (pomocí Telegraf agenta). Data byla pravidelně zapisována do databaze časových řad InfluxDB. Pro vizualizace dat byla použita platforma Grafana. U testovaných zařízeních byla provedena grafická vizualizace vytížení procesoru a jejich dostupnosti. Obě implementace byly testovány na síťových zařízeních Cisco c7200, Cisco IOSvL2, Juniper JunOS Olive a na serveru s operačním systémem Ubuntu 18.04. Na základě testování bylo zhodnoceno, že Nornir framework není vhodný pro automatizování serverů. Ačkoliv obě implementace splňují požadavky, které byly kladeny na automatizovaný systém, tak je nutno poznamenat, že vytvořený systém trpí nedostatky, které vycházejí zejména z omezených možností automatizace správy síťových zařízení,

které používají CLI a obrazů, které se používají zejména pro testovací účely. Na emulovaném zařízení Juniper Olive nelze například sbírat data o aktuálním vytížení hardwarových komponent. Dále nelze například zobrazit všechny statistiky týkající se přijatých a vysílaných paketů na jednotlivých rozhraní daného zařízení (funkcionalita není zcela implementována v NAPALM knihovně). V úplně poslední části práce byly obě implementace zhodnoceny a porovnány mezi sebou.

Implementované projekty lze dále rozšířit například o automatizovanou validaci síťové konfigurace pomocí tasku/modulu napalm_validate. Dále lze implementovaný systém rozšířit o automatizované testování Cisco zařízeních pomocí nástroje pyATS. Projekty lze dále vylepšit automatizací dalších konfigurací např. pro konfiguraci MPLS (Multiprotocol Label Switching) a BGP (Border Gateway Protocol) atd. V neposlední řadě lze navržený systém rozšířit o automatizaci konfigurace a monitorování síťových zařízení, které používají API.

Literatura

[1] CHOU, Eric. Mastering Python Networking. 3rd ed. Birmingham: Packt Publishing, 2020. ISBN 978-1-83921-467-7.

[2] BEN-KIKI, Oren, Clark EVANS a Ingy DÖT NET. YAML Ain't Markup Language (YAML™) Version 1.2. The Official YAML Web Site [online]. 2009 [cit. 2021-02-05]. Dostupné z: https://yaml.org/spec/1.2/spec.html

[3] EDELMAN, Jason, Scott S. LOWE a Matt OSWALT. Network Programmability and Automation. 1rd ed. Sebastopol: O'Reilly Media, c2018. ISBN 978-1-491-93125-7.

[4] What is YAML? ITPro. [online]. London: Dennis Publishing Limited, 2020 [cit. 2021-02-05]. Dostupné z: https://www.itpro.co.uk/development/34539/what-is-yaml

[5] BRAY, Tim, Ed. The JavaScript Object Notation (JSON) Data Interchange Format [online]. 2017 [cit. 2021-02-06]. RFC 8259. Dostupné z: https://tools.ietf.org/html/rfc8259

[6] Standard ECMA-404: The JSON Data Interchange Syntax [online]. 2rd ed. Geneva: Ecma International, 2017 [cit. 2021-02-06]. Dostupné z: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf

[7] Extensible Markup Language (XML) 1.1 (Second Edition). W3C [online]. W3C, 2006 [cit. 2021-02-06]. Dostupné z: https://www.w3.org/TR/xml11/

[8] HOLLENBECK, S., M. ROSE a L. MASINTER. Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols [online]. The Internet Society, 2003 [cit. 2021-02-06]. RFC 3470. Dostupné z: https://tools.ietf.org/html/rfc3470

[9] XML - Overview. Tutorialspoint [online]. Hyderabad, c2021 [cit. 2021-02-06]. Dostupné z: https://www.tutorialspoint.com/xml/xml_overview.htm

[10] XML-RPC - Introduction. Tutorialspoint [online]. Tutorialspoint, c2021 [cit. 2021-02-19]. Dostupné z: https://www.tutorialspoint.com/xml-rpc/xml_rpc_intro.htm

[11] ERP Tech Talk #2: XML-RPC vs JSON-RPC for Odoo. Medium [online]. Hummam Abdurrahim, 2020 [cit. 2021-02-19]. Dostupné z: https://medium.com/it-paragon/erp-tech-talk-2-xml-rpc-vs-json-rpc-for-odoo-f3daf6ae885

[12] JSON-RPC 2.0 Specification. JSON-RPC [online]. JSON-RPC Working Group, 2013 [cit. 2021-02-19]. Dostupné z: https://www.jsonrpc.org/specification

[13] Types of APIs. The Last Call - RapidAPI Blog [online]. RapidAPI, 2021 [cit. 2021-02-19]. Dostupné z: https://rapidapi.com/blog/types-of-apis/

[14] SOAP Version 1.2 Part 0: Primer (Second Edition). World Wide Web Consortium [online]. W3C®, 2007 [cit. 2021-02-20]. Dostupné z: https://www.w3.org/TR/soap12-part0/

[15] SOAP Web Services Tutorial: What is SOAP Protocol? EXAMPLE. Guru99 [online]. Guru99, c2021 [cit. 2021-02-20]. Dostupné z: https://www.guru99.com/soap-simple-object-access-protocol.html

[16] What is REST. REST API Tutorial [online]. restfulapi.net, c2020 [cit. 2021-02-20]. Dostupné z: https://restfulapi.net/

[17] What is a REST API? MulesSoft [online]. MulesSoft, c2021 [cit. 2021-02-20]. Dostupné z: https://www.mulesoft.com/resources/api/what-is-rest-api-design

[18] SOAP vs. REST: A Look at Two Different API Styles. Upwork [online]. Upwork Global, 2017 [cit. 2021-02-20]. Dostupné z: https://www.upwork.com/resources/soap-vs-rest-a-look-at-two-different-api-styles

[19] gRPC. Docs.microsoft.com [online]. Microsoft, 2021 [cit. 2021-03-02]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc

[20] SANTOS, Omar. Software-Defined Networking Security and Network Programmability. Cisco [online]. Hoboken: Cisco Press, 2020 [cit. 2021-02-21]. Dostupné z: https://www.ciscopress.com/articles/article.asp?p=3004581&seqNum=3

[21] ENNS, R., M. BJORKLUND, J. SCHOENWAELDER a A. BIERMAN. Network Configuration Protocol (NETCONF). IETF Tools [online]. Internet Engineering Task Force, 2011 [cit. 2021-02-21]. RFC6241. Dostupné z: https://tools.ietf.org/html/rfc6241

[22] ALBRECHT, Matt. Introduction to NETCONF and Juniper YANG Models. Ultra Config Generator [online]. Ultra Config, 2019 [cit. 2021-02-21]. Dostupné z: https://ultraconfig.com.au/blog/introduction-to-netconf-and-juniper-yang-models/

[23] BIERMAN, A., M. BJORKLUND a K. WATSEN. RESTCONF Protocol. IETF Tools [online]. Internet Engineering Task Force, 2017 [cit. 2021-02-21]. RFC8040. Dostupné z: https://tools.ietf.org/html/rfc8040

[24] Device Programmability. NetworkLessons.com [online]. NetworkLessons.com, c2013-2021 [cit. 2021-02-21]. Dostupné z: https://networklessons.com/cisco/evolving-technologies/device-programmability

[25] XML Schema Tutorial - Part 1. Liquid Technologies [online]. Bradford: Liquid Technologies Limited, c2001-2021 [cit. 2021-02-08]. Dostupné z: https://www.liquid-technologies.com/xml-schema-tutorial/xsd-elements-attributes

[26] JSON Schema. REST API Tutorial [online]. restfulapi, c2020 [cit. 2021-02-08]. Dostupné z: https://restfulapi.net/json-schema/

[27] What is XML Schema (XSD)? Microsoft [online]. Microsoft, c2021, 2016 [cit. 2021-02-08]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms765537(v=vs.85)

[28] XML Schema Tutorial - Part 1. Liquid Technologies [online]. Bradford: Liquid Technologies Limited, c2001-2021 [cit. 2021-02-08]. Dostupné z: https://www.liquid-technologies.com/xml-schema-tutorial/xsd-elements-attributes

[29] BJORKLUND, M. The YANG 1.1 Data Modeling Language. IETF Tools [online]. Internet Engineering Task Force, 2016 [cit. 2021-02-09]. RFC7950. Dostupné z: https://tools.ietf.org/html/rfc7950

[30] YANG Opensource Tools for Data Modeling-driven Management. Cisco Blogs [online]. Benoit Claise, 2017 [cit. 2021-02-09]. Dostupné z: https://blogs.cisco.com/getyourbuildon/yang-opensource-tools-for-data-modeling-driven-management

[31] ALBRECHT, Matt. Learn YANG - Full Tutorial for Beginners. Ultra Config Generator [online]. Melbourne: Ultra Config, c2017 [cit. 2021-02-09]. Dostupné z: https://ultraconfig.com.au/blog/learn-yang-full-tutorial-for-beginners/

[32] RATAN, Abhishek. Practical Network Automation. 2rd ed. Birmingham: Packt Publishing, c2018. ISBN 978-1-78995-565-1.

[33] FREEMAN, James a Jesse KEATING. Mastering Ansible. 3rd ed. Birmingham: Packt Publishing, c2019. ISBN 978-1-78995-154-7.

[34] OTIENO, John. What is the difference between Paramiko and Netmiko? LinuxHint [online]. Morgan Hill: Linux Hint, 2021 [cit. 2021-02-14]. Dostupné z: https://linuxhint.com/paramiko-difference-netmiko/

[35] ALBRECHT, Matt. NETMIKO Automation on Juniper Full Tutorial for Beginners. Ultra Config Generator [online]. Melbourne: Ultra Config, 2020 [cit. 2021-02-14]. Dostupné z: https://ultraconfig.com.au/blog/netmiko-automation-on-juniper-routers-full-tutorial-for-beginners/

[36] Supported Devices. NAPALM [online]. David Barroso, c2020 [cit. 2021-02-14]. Dostupné z: https://napalm.readthedocs.io/en/latest/support/index.html

[37] Network Automation Using Unified API – Napalm. Cisco Blogs [online]. Stuart Clark, 2019 [cit. 2021-02-14]. Dostupné z: https://blogs.cisco.com/developer/network-automation-using-napalm¨

[38] Template Designer Documentation. Jinja [online]. Pallets, c2007 [cit. 2021-02-14]. Dostupné z: https://jinja.palletsprojects.com/en/2.11.x/templates/

[39] Basic Concepts. Ansible Documentation [online]. Red Hat, 2021 [cit. 2021-02-16]. Dostupné z: https://docs.ansible.com/ansible/latest/network/getting_started/basic_concepts.html

[40] HOCHSTEIN, Lorin a René MOSER. Ansible Up & Running. 2rd ed. Sebastopol: O'Reilly Media, c2017. ISBN 978-1-491-97980-8.

[41] OGENSTAD, Patrick. Ansible vs. Nornir: Speed Challenge. Networklore [online]. Networklore, 2019 [cit. 2021-02-15]. Dostupné z: https://networklore.com/ansible-nornir-speed/

[42] OGENSTAD, Patrick. Introducing Nornir - The Python automation framework. Networklore [online]. Networklore, 2018 [cit. 2021-02-15]. Dostupné z: https://networklore.com/introducing-brigade/

[43] Nornir [online]. David Barroso, c2020 [cit. 2021-02-15]. Dostupné z: https://nornir.readthedocs.io/en/latest/index.html

[44] Nornir Plugins. Nornir.tech [online]. Djordje Atlialp, c2021 [cit. 2021-02-15]. Dostupné z: https://nornir.tech/nornir/plugins/

[45] Plugins. Nornir [online]. David Barroso, c2020 [cit. 2021-02-15]. Dostupnéz:https://nornir.readthedocs.io/en/latest/plugins/index.html

[46] Nornir. In: Said van de Klundert [online]. Said van de Klundert, 2020 [cit. 2021-02-15]. Dostupné z: https://saidvandeklundert.net/2020-12-06-nornir/

[47] PAOLO BOARINA, Gian. Using IP Fabric with Nornir. IP FABRIC [online]. IP Fabric, 2020 [cit. 2021-02-15]. Dostupné z: https://ipfabric.io/blog/using-ip-fabric-with-nornir/

[48] Introduction to StackStorm. Matt Oswalt [online]. Matt Oswalt, c2010-2021 [cit. 2021-02-17]. Dostupné z: https://oswalt.dev/2016/12/introduction-to-stackstorm/

[49] StackStorm Overview. StackStorm Documentation [online]. StackStorm, c2014-2020 [cit. 2021-02-17]. Dostupné z: https://docs.stackstorm.com/overview.html#how-it-works

[50] Chef Tutorial for Beginners. Tutorial And Example [online]. Tutorial And Example, 2019 [cit. 2021-02-17]. Dostupné z: https://www.tutorialandexample.com/chef-tutorial/

[51] What is Chef? Chef Tutorial Guide for Beginner. JanBask Training [online]. Janbask Training, 2020 [cit. 2021-02-17]. Dostupné z: https://www.janbasktraining.com/blog/chef-tutorial/

[52] What is version control? Atlassian [online]. Sydney: Atlassian, c2021 [cit. 2021-01-31]. Dostupné z: https://www.atlassian.com/git/tutorials/what-is-version-control

[53] CHACON, Scott a Ben STRAUB. Pro Git [online]. 2rd ed. New York: Springer, c2014 [cit. 2021-01-31]. ISBN 978-1-4842-0076-6. Dostupné z: https://git-scm.com/book/en/v2

[54] What is streaming telemetry? Blue Planet [online]. Don Jacob, 2018 [cit. 2021-03-02]. Dostupné z: https://www.blueplanet.com/blog/What-is-streaming-telemetry.html

[55] KEARY, Tim. SNMP Protocol Architecture – MIBs and OIDs explained. ITPRC [online]. ITPRC, 2020 [cit. 2021-02-23]. Dostupné z: https://www.itprc.com/snmp-protocol-architecture-mibs-oids/

[56] ELLINGWOOD, Justin. An Introduction to SNMP (Simple Network Management Protocol). DigitalOcean Community Home [online]. DigitalOcean, 2014 [cit. 2021-02-23]. Dostupné z: https://www.digitalocean.com/community/tutorials/an-introduction-to-snmp-simple-network-management-protocol

[57] KEARY, Tim. What is SNMP? A Simple Network Management Protocol Tutorial. ITPRC [online]. ITPRC, 2018 [cit. 2021-03-01]. Dostupné z: https://www.itprc.com/what-is-snmp/

[58] Model-Driven Telemetry. Programmability Configuration Guide, Cisco IOS XE Everest 16.6.x [online]. San Jose: Cisco Systems, 2017, s. 117-119 [cit. 2021-03-02]. Dostupné z: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/prog/configuration/166/b_166_programmability_cg.pdf

[59] ALBRECHT, Matt. Cisco Model-Driven Telemetry tutorial with Telegraf, InfluxDB, and Grafana. Ultra Config Generator [online]. Ultra Config, 2020 [cit. 2021-03-02]. Dostupné z: https://ultraconfig.com.au/blog/cisco-telemetry-tutorial-with-telegraf-influxdb-and-grafana/

[60] Introduction to the Fundamentals of Time Series Data and Analysis. Aptech [online]. Arizona: Aptech, 2018 [cit. 2021-02-26]. Dostupné z: https://www.aptech.com/blog/introduction-to-the-fundamentals-of-time-series-data-and-analysis/

[61] Introduction to time series. Grafana: The open observability platform [online]. Grafana Labs, c2021 [cit. 2021-02-26]. Dostupné z: https://grafana.com/docs/grafana/latest/getting-started/timeseries/

[62] 7 Powerful Time-Series Database for Monitoring Solution. Geekflare - Technical Articles, Tools and Awesome Resources [online]. Avi, 2020 [cit. 2021-02-28]. Dostupné z: https://geekflare.com/time-series-database/

[63] FLORY, Justin W. How time-series databases help make sense of sensors. Opensource.com [online]. Red Hat, 2017 [cit. 2021-02-28]. Dostupné https://opensource.com/article/17/8/influxdb-time-series-database-stack

[64] TimescaleDB vs. InfluxDB: Purpose built differently for time-series data. Timescale [online]. Timescale, 2020 [cit. 2021-02-28]. Dostupné z: https://blog.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299

[65] Line protocol. InfluxData Documentation [online]. InfluxData, c2021 [cit. 2021-02-28]. Dostupné z: https://docs.influxdata.com/influxdb/cloud/reference/syntax/line-protocol/

[66] HBase - Overview. Tutorialspoint [online]. Tutorialspoint, c2021 [cit. 2021-02-27]. Dostupné z: https://www.tutorialspoint.com/hbase/hbase_overview.htm

[67] How does OpenTSDB work? OpenTSDB [online]. The OpenTSDB Authors, c2010-2021 [cit. 2021-02-27]. Dostupné z: http://opentsdb.net/overview.html

[68] TimescaleDB 101: the why, what, and how. Aiven - Medium [online]. Aiven, 2018 [cit. 2021-02-27]. Dostupné z: https://aiven-io.medium.com/timescaledb-101-the-why-what-and-how-9c0eb08a7c0b

[69] OLES, Bart. An Introduction to TimescaleDB. Severalnines [online]. Kalmar: severalnines, 2019 [cit. 2021-02-27]. Dostupné z: https://severalnines.com/database-blog/introduction-timescaledb

[70] CAMPION, Nick. Getting Started with Grafana. MetricFire [online]. MetricFire Corporation, 2020 [cit. 2021-02-28]. Dostupné z: https://www.metricfire.com/blog/getting-started-with-grafana/

[71] PATIL, Harshal. Introduction to Grafana. Aspire Software Solutions [online]. Gujurat: Aspire Software Solutions, 2020 [cit. 2021-02-28]. Dostupné z: https://aspiresoftware.in/blog/introduction-to-grafana/

[72] Telegraf. InfluxDB: Purpose-Built Open Source Time Series Database [online]. San Francisco: InfluxData, c2021 [cit. 2021-02-25]. Dostupné z: https://www.influxdata.com/time-series-platform/telegraf/

[73] HAGAN, Jim. How to Use Custom Telemetry From Telegraf in New Relic One. New Relic [online]. New Relic, 2020 [cit. 2021-02-25]. Dostupné z: https://blog.newrelic.com/engineering/how-to-collect-telegraf-metrics/

[74] Telegraf plugins. InfluxData Documentation [online]. InfluxData, [2021] [cit. 2021-02-25]. Dostupné z: https://docs.influxdata.com/telegraf/v1.17/plugins/

[75] The Definitive Comparison Guide: Nagios XI vs Nagios Core. Nagios [online]. Saint Paul: Nagios Team, 2021 [cit. 2021-02-25]. Dostupné z: https://www.nagios.com/news/2021/01/comparison-guide-nagios-xi-vs-nagios-core/

[76] KOCJAN, Wojciech a Piotr BELTOWSKI. Learning Nagios Third Edition [online]. 3rd ed. Birmingham: Packt Publishing, 2016 [cit. 2021-02-25]. ISBN 978-1-78588-595-2. Dostupné z: https://oiipdf.cdn.oii.ink/pdf/fa81185e-3b89-4f02-adf8-15f5240a1508.pdf

[77] Nagios - Architecture. Tutorialspoint [online]. Hyderabad: Tutorialspoint, c2021 [cit. 2021-02-25]. Dostupné z: https://www.tutorialspoint.com/nagios/nagios_architecture.htm

[78] NDOUTILS Documentation Version 2.x [online]. 2rd ed. Ethan Galstad, 2017 [cit. 2021-02-25]. Dostupné z: https://assets.nagios.com/downloads/nagioscore/docs/ndoutils/NDOUtils.pdf

[79] Introduction to Zabbix. Tim's Place [online]. Hong Kong: Tim Kehres, c2003-2020 [cit. 2021-02-24]. Dostupné z: https://tim.kehres.com/zabbix/intro/

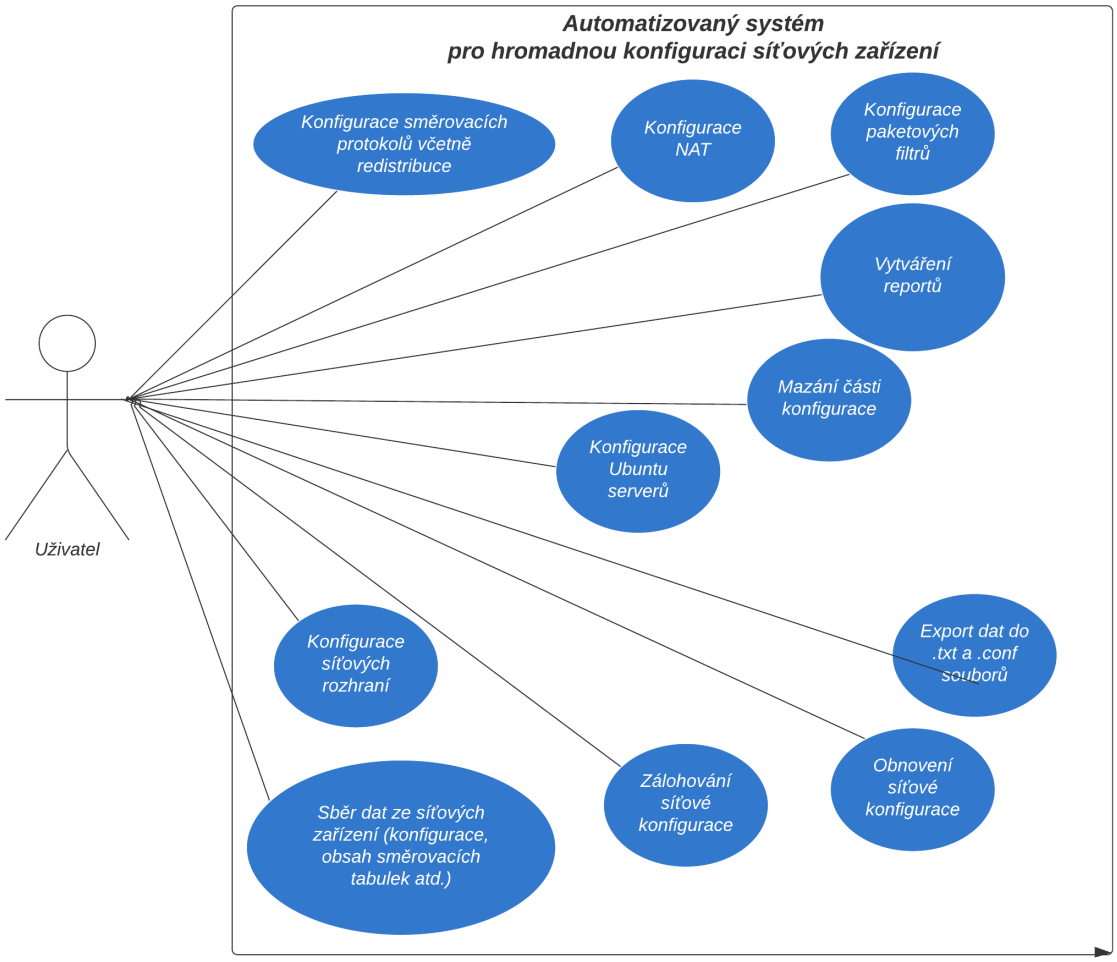
[80] ZABBIX OVERVIEW. Zabbix Documentation 5.2 [online]. Zabbix, c2001-2021 [cit. 2021-02-24]. Dostupné z: https://www.zabbix.com/documentation/current/manual/introduction/overview

[81] Monitor your Infrastructure with TIG Stack. Hacker Noon [online]. Mohamed Labouardy, 2017 [cit. 2021-03-01]. Dostupné z: https://hackernoon.com/monitor-your-infrastructure-with-tig-stack-b63971a15ccf

[82] How To Setup Telegraf InfluxDB and Grafana on Linux. Devconnected [online]. devconnected, c2021 [cit. 2021-03-01]. Dostupné z: https://devconnected.com/how-to-setup-telegraf-influxdb-and-grafana-on-linux/

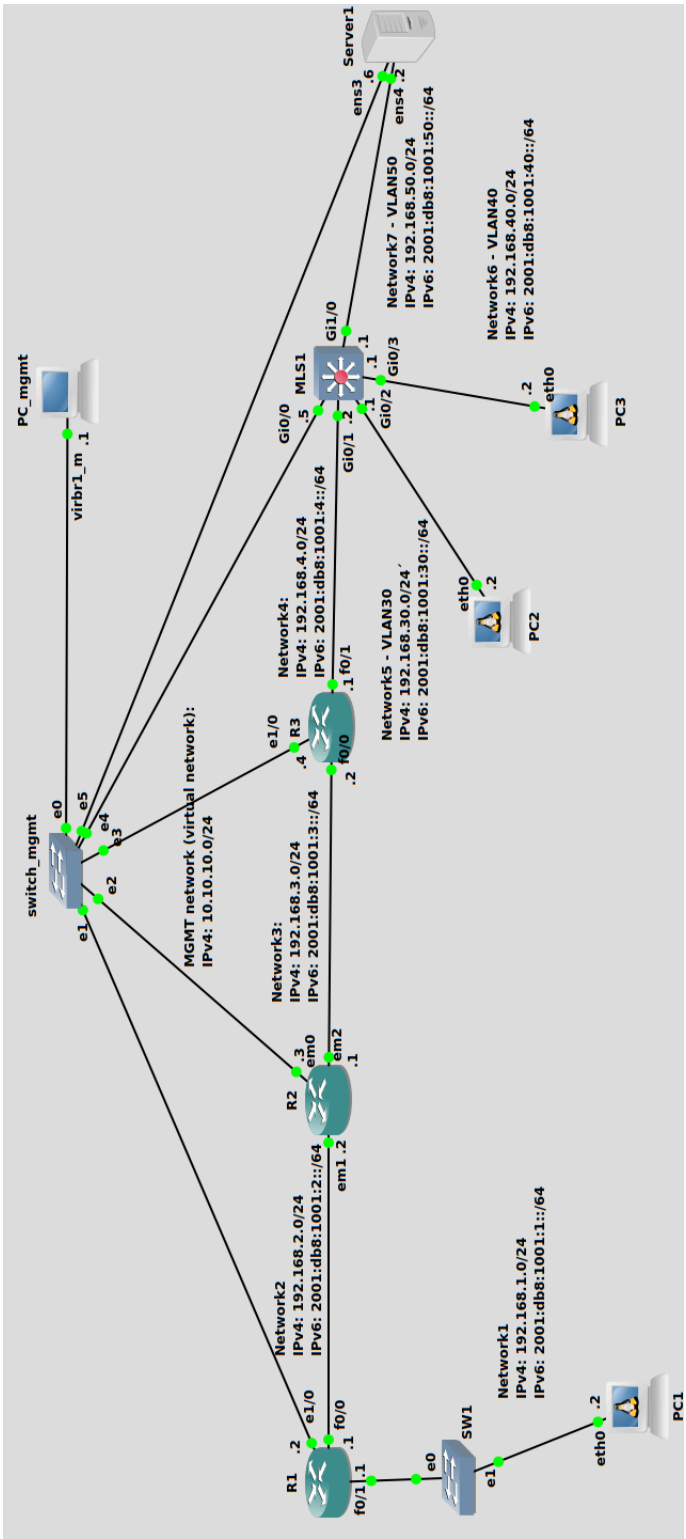
Seznam příloh

A UML diagram případů užití části automatizovaného systému pro konfiguraci síťových zařízení



Obrázek 39: UML diagram případů užití automatizovaného systému pro hromadnou konfiguraci síťových zařízení

B Síťová topologie



Obrázek 40: Síťová topologie pro testování navržených řešení

C Prvotní konfigurace síťových zařízení

```
//Povolit směrování IPv6 unicast provozu
R1(config)#ipv6 unicast-routing

//Zákáz AAA autentifikace, povolení SCP serveru, nastavení doménového jména
R1(config)#no aaa new-model
R1(config)#ip scp server enable
R1(config)#ip domain-name automation.local

//Vytvoření privilegovaného uživatele admin
R1(config)#enable secret cisco
R1(config)#username admin privilege 15 secret automationDP

//Nastavení archivace konfigurace pro rollback (nutné u NAPALM replace metody)
R1(config)#archive
R1(config-archive)#path disk0:
R1(config-archive)#write-memory

//Konfigurace síťového rozhraní
R1(config)#int e1/0
R1(config-if)#description e1/0 connected to switch_mgmt e1
R1(config-if)#ip address 10.10.10.2 255.255.255.0
R1(config-if)#duplex full
R1(config-if)#no shutdown

//Konfigurace virtuálního terminálu
R1(config)#line vty 0 4
R1(config-line)#transport input ssh
R1(config-line)#login local

//Generování asymetrických klíčů (2048bitové RSA klíče)
R1(config)#crypto key generate rsa
The name for the keys will be: R1.automation.local
Choose the size of the key modulus in the range of 360 to 4096 for your
General Purpose Keys. Choosing a key modulus greater than 512 may take
a few minutes.

How many bits in the modulus [512]: 2048
```

```
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 6 seconds)

R1(config)#ip ssh version 2
R1(config)#ip domain lookup
```

Výpis 11: Počáteční konfigurace Cisco směrovače R1 (s komentáři)

```
[edit]
root# set system root-authentication plain-text-password
New password: Automation
Retype new password: Automation

[edit]
root# set system host-name R2
root# set system domain-name automation.local
root# set interfaces em0 unit 0 family inet address 10.10.10.3/24
root# set interfaces em0 description "e1/0 connected to switch_mgmt e2"
root# set system services ssh
root# set system services ssh root-login deny
root# set system login user juniper class super-user
root# set system login user juniper authentication plain-text-password
New password: Juniper
Retype new password: Juniper
root# commit
```

Výpis 12: Počáteční konfigurace Juniper směrovače R2

```
//Povolit směrování IPv6 unicast provozu
R3(config)#ipv6 unicast-routing

//Zákaz AAA autentifikace, povolení SCP serveru, nastavení doménového jména
R3(config)#no aaa new-model
R3(config)#ip scp server enable
R3(config)#ip domain-name automation.local

//Vytvoření privilegovaného uživatele admin
R3(config)#enable secret cisco
R3(config)#username admin privilege 15 secret automationDP

//Nastavení archivace konfigurace pro rollback (nutné u NAPALM replace metody)
```

```
R3(config)#archive
R3(config-archive)#path disk0:
R3(config-archive)#write-memory

//Konfigurace síťového rozhraní
R3(config)#int e1/0
R3(config-if)#description e1/0 connected to switch_mgmt e3
R3(config-if)#ip address 10.10.10.4 255.255.255.0
R3(config-if)#duplex full
R3(config-if)#no shutdown

//Konfigurace virtuálního terminálu
R3(config)#line vty 0 4
R3(config-line)#transport input ssh
R3(config-line)#login local

//Generování asymetrických klíčů (2048bitové RSA klíče)
R3(config)#crypto key generate rsa
The name for the keys will be: R3.automation.local
Choose the size of the key modulus in the range of 360 to 4096 for your
General Purpose Keys. Choosing a key modulus greater than 512 may take
a few minutes.

How many bits in the modulus [512]: 2048
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 6 seconds)

R3(config)#ip ssh version 2
R3(config)#ip domain lookup
```

Výpis 13: Počáteční konfigurace Cisco směrovače R3 (s komentáři)

```
Switch(config)#hostname MLS1
//Povolit směrování IPv4 paketů a směrování IPv6 unicast provozu
MLS1(config)#ip routing
MLS1(config)#ipv6 unicast-routing

//Konfigurace VTP v transparentním módu, zakáz zobrazení bannerů
MLS1(config)#vtp mode transparent
MLS1(config)#no banner incoming
```

```
MLS1(config)#no banner login
MLS1(config)#no banner exec

//Zákáz AAA autentifikace, povolení SCP serveru, nastavení doménového jména
MLS1(config)#no aaa new-model
MLS1(config)#ip scp server enable
MLS1(config)#ip domain-name automation.local

//Vytvoření privilegovaného uživatele admin
MLS1(config)#enable secret cisco
MLS1(config)#username admin privilege 15 secret automationDP

//Nastavení archivace konfigurace pro rollback (nutné u NAPALM replace metody)
MLS1(config)#archive
MLS1(config-archive)#path disk0:
MLS1(config-archive)#write-memory

//Konfigurace síťového rozhraní
MLS1(config)#int g0/0
MLS1(config-if)#description g0/0 connected to switch_mgmt e4
MLS1(config-if)#ip address 10.10.10.5 255.255.255.0
MLS1(config-if)#no negotiation auto
MLS1(config-if)#duplex full
MLS1(config-if)#no shutdown

//Konfigurace virtuálního terminálu
MLS1(config)#line vty 0 4
MLS1(config-line)#transport input ssh
MLS1(config-line)#login local

//Generování asymetrických klíčů (2048bitové RSA klíče)
MLS1(config)#crypto key generate rsa
The name for the keys will be: MLS1.automation.local
Choose the size of the key modulus in the range of 360 to 4096 for your
General Purpose Keys. Choosing a key modulus greater than 512 may take
a few minutes.

How many bits in the modulus [512]: 2048
% Generating 2048 bit RSA keys, keys will be non-exportable...
```

```
[OK] (elapsed time was 6 seconds)

MLS1(config)#ip ssh version 2
MLS1(config)#ip domain lookup
```

Výpis 14: Počáteční konfigurace L3 switche MLS1 (s komentáři)

```
network:
   version: 2
   renderer: networkd
   ethernets:
      ens3:
         dhcp4: no
         addresses:
          - 10.10.10.6/24
         gateway4: 10.10.10.1
         nameservers:
            addresses: [8.8.4.4,8.8.8.8]
      ens4:
         accept-ra: no
         addresses:
          - 192.168.50.2/24
          - 2001:db8:1001:50::2/64
```

Výpis 15: Konfigurace síťových rozhraní zařízení Server1

```
auto eth0
iface eth0 inet static
   address 192.168.1.2
   netmask 255.255.255.0
   gateway 192.168.1.1

iface eth0 inet6 static
   address 2001:db8:1001:1::2
   netmask 64
   gateway 2001:db8:1001:1::1
```

Výpis 16: Konfigurace síťových rozhraní zařízení PC1

```
auto eth0
iface eth0 inet static
```

```
   address 192.168.30.2
   netmask 255.255.255.0
   gateway 192.168.30.1

iface eth0 inet6 static
   address 2001:db8:1001:30::2
   netmask 64
   gateway 2001:db8:1001:30::1
```

Výpis 17: Konfigurace síťových rozhraní zařízení PC2

```
auto eth0
iface eth0 inet static
   address 192.168.40.2
   netmask 255.255.255.0
   gateway 192.168.40.1

iface eth0 inet6 static
   address 2001:db8:1001:40::2
   netmask 64
   gateway 2001:db8:1001:40::1
```

Výpis 18: Konfigurace síťových rozhraní zařízení PC3

D Pipfile soubory

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
napalm-ansible = "*"
napalm = "*"
ansible = "*"
influxdb = "*"

[dev-packages]

[requires]
python_version = "3.8"
```

Výpis 19: Obsah Pipfile souboru Ansible projektu

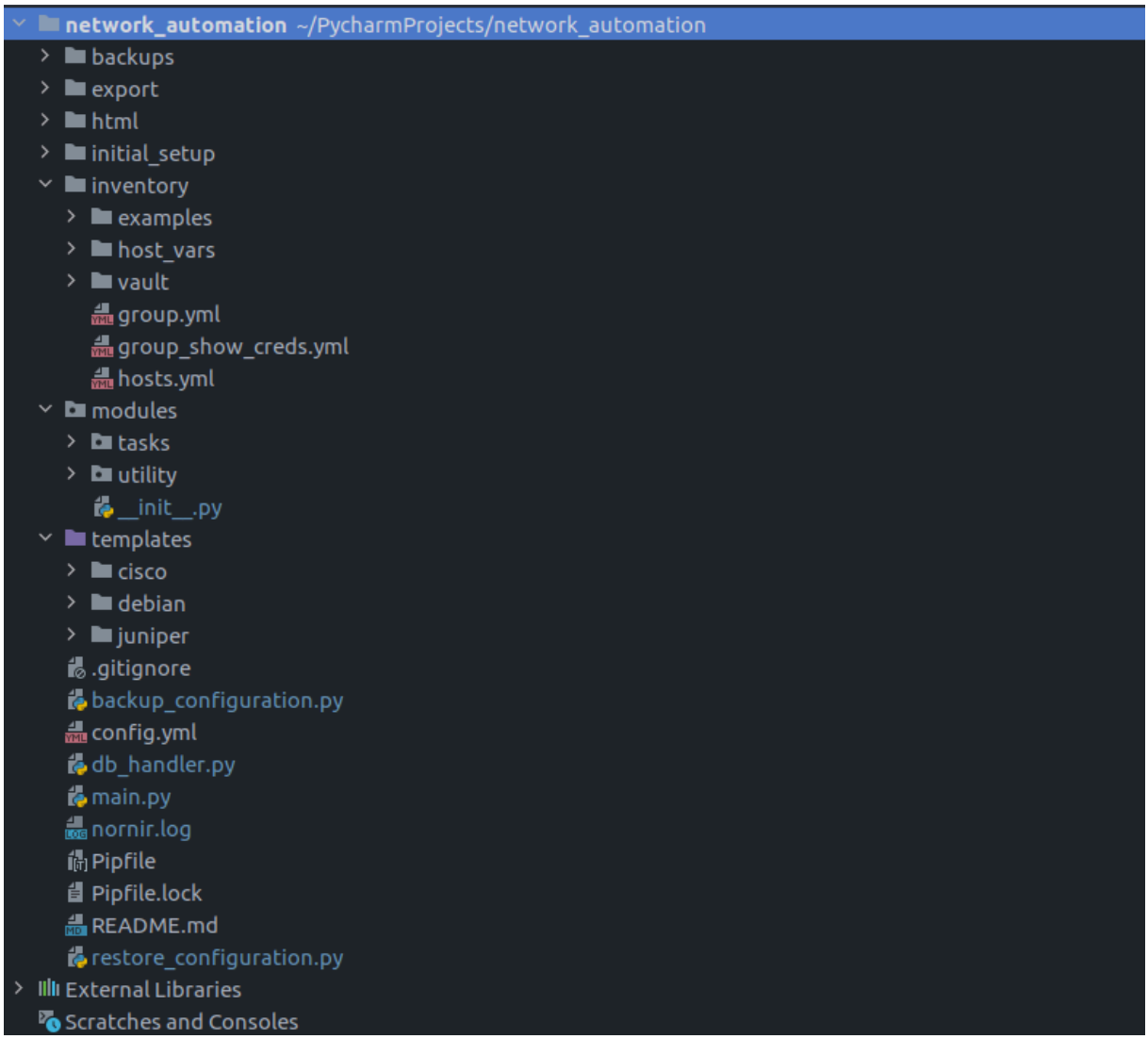
```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true
[dev-packages]

[packages]
Jinja2 = "*"
PyYAML = "*"
textfsm = "*"
pytest = "*"
setuptools = "*"
simplejson = "*"
yamllint = "*"
requests = "*"
pylint = "*"
lxml = "*"
ntc-templates = "*"
cryptography = "*"
junos-eznc = "*"
```

```
nornir = "*"
nornir-napalm = "*"
nornir-netmiko = "*"
nornir-ansible = "*"
nornir-utils = "*"
nornir-jinja2 = "*"
prettytable = "*"
openpyxl = "*"
pdoc3 = "*"
jinja2 = "*"
ansible = "*"
influxdb = "*"

[requires]
python_version = "3.8"
```

Výpis 20: Obsah Pipfile souboru Nornir projektu

E Struktura projektů

Obrázek 41: Struktura Ansible projektu



Obrázek 42: Struktura Nornir projektu

F Inventář - Ansible projekt

```
[cisco_l3_switches]
MLS1

[cisco_routers]
R1
R3

[cisco_l3:children]
cisco_routers
cisco_l3_switches

[cisco:children]
cisco_l3

[juniper_olive]
R2

[juniper_routers:children]
juniper_olive

[juniper:children]
juniper_routers

[routers:children]
cisco_routers
juniper_routers

[network_devices:children]
cisco
juniper

[ubuntu_servers]
Server1

[debian:children]
ubuntu_servers
```

```
[linux:children]
debian
```

Výpis 21: Obsah souboru hosts.ini

```
---
ansible_host: 10.10.10.2
vendor: cisco
dev_type: router
image: "c7200"
interfaces_ipv4:
   FastEthernet0/0:
      description: connected to R2 on port em1
      net: 192.168.2.1
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to SW1 on port e0
      net: 192.168.1.1
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
      speed: auto
ospf_config:
   process: 1
   router_id: 1.1.1.1
   passive_interfaces: [ FastEthernet0/1 ]
   networks:
      - ip: 192.168.1.0
        wildcard: 0.0.0.255
        area_number: 0
      - ip: 192.168.2.0
        wildcard: 0.0.0.255
        area_number: 0
interfaces_ipv6:
   FastEthernet0/0:
      description: connected to R2 on port em1
      networks:
```

```
        - net: "2001:db8:1001:2::1"
          prefix_length: /64
          eui_format_auto: false
      routed_physical_port: true
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to SW1 on port e0
      networks:
         - net: "2001:db8:1001:1::1"
           prefix_length: /64
           eui_format_auto: false
      routed_physical_port: true
      duplex: full
      speed: auto
ospfv3_config:
   process: 1
   router_id: 1.1.1.1
   passive_interfaces: [ FastEthernet0/1 ]
   interfaces:
      - name: FastEthernet0/0
        area_number: 0
      - name: FastEthernet0/1
        area_number: 0
restore_config:
   running_config_date: 2021-04-07
delete_config:
   interfaces:
      physical: [FastEthernet0/0, FastEthernet0/1]
   ospf_config:
      processes: [1]
   ospfv3_config:
      processes:
         - number: 1
           interfaces:
            - name: FastEthernet0/0
              area_number: 0
            - name: FastEthernet0/1
              area_number: 0
```

Výpis 22: Obsah souboru R1.yml

```
---
ansible_host: 10.10.10.4
vendor: cisco
dev_type: router
image: "c7200"
interfaces_ipv4:
   FastEthernet0/0:
      description: connected to R2 on port em2
      net: 192.168.3.2
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to MLS1 on port g0/1
      net: 192.168.4.1
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
      speed: auto
ospf_config:
   process: 1
   router_id: 3.3.3.3
   networks:
      - ip: 192.168.3.0
        wildcard: 0.0.0.255
        area_number: 0
   redistribute:
      static: false
      default: false
      eigrp:
        AS_numbers: [1]
eigrp_config:
   AS_number: 1
   passive_interfaces: []
   networks:
```

```
      - ip: 192.168.4.0
        wildcard: 0.0.0.255
   redistribute:
      ospf:
         processes:
            - number: 1
              bandwidth: 100000
              delay: 1
              reliability: 255
              load: 1
              mtu: 1500
interfaces_ipv6:
   FastEthernet0/0:
      description: connected to R2 on port em2
      networks:
         - net: "2001:db8:1001:3::2"
           prefix_length: /64
           eui_format_auto: false
      routed_physical_port: true
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to MLS1 on port g0/1
      networks:
         - net: "2001:db8:1001:4::1"
           prefix_length: /64
           eui_format_auto: false
      routed_physical_port: true
      duplex: full
      speed: auto
ospfv3_config:
   process: 1
   router_id: 3.3.3.3
   interfaces:
      - name: FastEthernet0/0
        area_number: 0
   redistribute:
      static: false
      default: false
```

```
      eigrp:
         AS_numbers: [1]
restore_config:
   running_config_date: 2021-04-07
eigrp_ipv6_config:
   AS_number: 1
   router_id: 3.3.3.3
   passive_interfaces: []
   interfaces: [ FastEthernet0/1 ]
   redistribute:
      ospf:
         processes:
            - number: 1
              bandwidth: 100000
              delay: 1
              reliability: 255
              load: 1
              mtu: 1500
delete_config:
   interfaces:
      physical: [FastEthernet0/0, FastEthernet0/1]
   eigrp_config:
      AS_numbers: [ 1 ]
   eigrp_ipv6_config:
      AS:
         - number: 1
           interfaces: [ FastEthernet0/1 ]
   ospf_config:
      processes: [1]
   ospfv3_config:
      processes:
         - number: 1
            interfaces:
               - name: FastEthernet0/0
                 area_number: 0
```

Výpis 23: Obsah souboru R3.yml

```
---
ansible_host: 10.10.10.5
```

```
vendor: cisco
dev_type: L3_switch
image: "IOSvL2"
vlans_config:
   - number: 30
     name: VLAN30
   - number: 40
     name: VLAN40
   - number: 50
     name: VLAN50
switching_interfaces:
   GigabitEthernet0/2:
      description: connected to PC2 on port eth0
      mode: access
      vlan: 30
   GigabitEthernet0/3:
      description: connected to PC3 on port eth0
      mode: access
      vlan: 40
   GigabitEthernet1/0:
      description: connected to Server1 on port ens4
      mode: access
      vlan: 50
interfaces_ipv4:
   GigabitEthernet0/1:
      description: connected to R3 on port f0/1
      net: 192.168.4.2
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
   Vlan30:
      description: VLAN30 SVI
      net: 192.168.30.1
      mask: 255.255.255.0
   Vlan40:
      description: VLAN40 SVI
      net: 192.168.40.1
      mask: 255.255.255.0
   Vlan50:
```

```
      description: VLAN50 SVI
      net: 192.168.50.1
      mask: 255.255.255.0
eigrp_config:
   AS_number: 1
   passive_interfaces: []
   networks:
   - ip: 192.168.4.0
     wildcard: 0.0.0.255
   - ip: 192.168.30.0
     wildcard: 0.0.0.255
   - ip: 192.168.40.0
     wildcard: 0.0.0.255
   - ip: 192.168.50.0
     wildcard: 0.0.0.255
packet_filter_config:
   VLAN30in:
   type: extended
   rules:
      - deny ip 192.168.30.0 0.0.0.255 192.168.40.0 0.0.0.255
      - permit ip any any
   routed_interfaces:
      - name: Vlan30
        acl_direction: in
   VLAN40in:
   type: extended
   rules:
      - permit ip 192.168.40.0 0.0.0.255 192.168.30.0 0.0.0.255
      - permit ip any any
   routed_interfaces:
      - name: Vlan40
        acl_direction: in
interfaces_ipv6:
   GigabitEthernet0/1:
      description: connected to R3 on port f0/1
      networks:
         - net: "2001:db8:1001:4::2"
           prefix_length: /64
           eui_format_auto: false
```

```
      routed_physical_port: true
      duplex: full
      speed: auto
   Vlan30:
      description: VLAN30 SVI
      networks:
         - net: "2001:db8:1001:30::1"
           prefix_length: /64
           eui_format_auto: false
   Vlan40:
      description: VLAN40 SVI
      networks:
         - net: "2001:db8:1001:40::1"
           prefix_length: /64
           eui_format_auto: false
   Vlan50:
      description: VLAN50 SVI
      networks:
         - net: "2001:db8:1001:50::1"
           prefix_length: /64
           eui_format_auto: false
eigrp_ipv6_config:
   AS_number: 1
   router_id: 4.4.4.4
   routed_interfaces: [ Vlan30, Vlan40, Vlan50, GigabitEthernet0/1 ]
packet_filter_ipv6_config:
   VLAN30-in:
      rules:
         - deny ipv6 2001:db8:1001:30::/64 2001:db8:1001:40::/64
         - permit ipv6 any any
      routed_interfaces:
         - name: Vlan30
           acl_direction: in
   VLAN40-in:
      rules:
         - deny ipv6 2001:db8:1001:40::/64 2001:db8:1001:30::/64
         - permit ipv6 any any
      routed_interfaces:
         - name: Vlan40
```

```
          acl_direction: in
restore_config:
   running_config_date: 2021-04-07
delete_config:
   interfaces:
      virtual: [Vlan30,Vlan40,Vlan50]
      physical: [GigabitEthernet0/1, GigabitEthernet0/2, GigabitEthernet0/3,
          GigabitEthernet1/0]
   eigrp_config:
      AS_numbers: [1]
   packet_filter_config:
      Vlan30in:
         type: extended
         routed_interfaces:
            - name: Vlan30
              acl_direction: in
      Vlan40in:
         type: extended
         routed_interfaces:
            - name: Vlan40
              acl_direction: in
   eigrp_ipv6_config:
      AS:
         - number: 1
           routed_interfaces: [ Vlan30,Vlan40,Vlan50, GigabitEthernet0/1 ]
   packet_filter_ipv6_config:
      VLAN30-in:
         routed_interfaces:
            - name: Vlan30
               acl_direction: in
      VLAN40-in:
         routed_interfaces:
            - name: Vlan40
              acl_direction: in
   vlans_config:
      numbers: [30, 40, 50]
```

Výpis 24: Obsah souboru MLS1.yml

```
---
```

```
ansible_host: 10.10.10.3
vendor: juniper
dev_type: router
image: "olive"
interfaces_ipv4:
   em1:
      description: '"connected to R1 on port f0/0"'
      units:
         - number: 0
           net: 192.168.2.2
           prefix_length: /24
   em2:
      description: '"connected to R3 on port f0/0"'
      units:
         - number: 0
           net: 192.168.3.1
           prefix_length: /24
ospf_config:
   router_id: 2.2.2.2
   interfaces:
      em1.0:
         passive: false
         area_number: 0
      em2.0:
         passive: false
         area_number: 0
interfaces_ipv6:
   em1:
      description: '"connected to R1 on port f0/0"'
      units:
         - number: 0
           net: "2001:db8:1001:2::2"
           prefix_length: /64
   em2:
      description: '"connected to R3 on port f0/0"'
      units:
         - number: 0
           net: "2001:db8:1001:3::1"
           prefix_length: /64
```

```
ospfv3_config:
   router_id: 2.2.2.2
   interfaces:
      em1.0:
         passive: false
         area_number: 0
      em2.0:
         passive: false
         area_number: 0
restore_config:
   running_config_date: 2021-04-07
delete_config:
   interfaces:
   em1:
      delete_whole_interface: true
      delete_units: []
   em2:
      delete_whole_interface: false
      delete_units: [0]
   ospf_config:
      router_id: 2.2.2.2
   ospfv3_config:
      router_id: 2.2.2.2
```

Výpis 25: Obsah souboru R2.yml

```
---
ansible_host: 10.10.10.6
vendor: debian
dev_type: ubuntu_server
vsftpd_config:
   vsftpd_user:
      name: ftpuser
      password: "{{ vault_vsftpd_user_password | string | password_hash('sha512
          ') }}"
      groups: sudo,ftpuser
      home: /home/ftpuser
   ssl:
      enabled: true
      cert_file: /etc/certs/vsftpd_cert.pem
```

```
      openssl_csr: /etc/ssl/private/vsftpd.csr
      private_key_file: /etc/certs/vsftpd.pem
      cn: "{{ ansible_host }}"

telegraf_ip_address: "{{ ansible_host }}"
influxdb_ip_address: "{{ ansible_host }}"
influx_db_url: "http://{{ ansible_host }}:8086"
grafana_url: "http://{{ ansible_host }}:3000"

grafana_data_sources:
   - name: influxsourceAnsible
     ds_type: "influxdb"
     database: "{{ influxdb_ansible_database_name }}"
     url: "{{ influx_db_url }}"
     user: "{{ influxdb_admin_username }}"
     password: "{{ influxdb_admin_password }}"
   - name: influxsourceNornir
     ds_type: "influxdb"
     database: "{{ influxdb_nornir_database_name }}"
     url: "{{ influx_db_url }}"
     user: "{{ influxdb_admin_username }}"
     password: "{{ influxdb_admin_password }}"
   - name: influxsourceTelegraf
     ds_type: "influxdb"
     database: "{{ influxdb_telegraf_database_name }}"
     url: "{{ influx_db_url }}"
     user: "{{ influxdb_admin_username }}"
     password: "{{ influxdb_admin_password }}"
```

Výpis 26: Obsah souboru vars u zařízení Server1

```
---
ansible_python_interpreter: ~/.local/share/virtualenvs/
    network_automation_ansible-jyRQ_HBH/bin/python
ansible_network_os: ios
ansible_connection: network_cli
ansible_user: admin
ansible_password: "{{ vault_ansible_password }}"
napalm_provider:
   optional_args: { 'secret': "{{ vault_secret }}", 'global_delay_factor': 2 }
```

```
  hostname: "{{ ansible_host }}"
  username: "{{ ansible_user }}"
  dev_os: "{{ ansible_network_os }}"
  password: "{{ ansible_password }}"
```

Výpis 27: Obsah souboru vars skupiny cisco

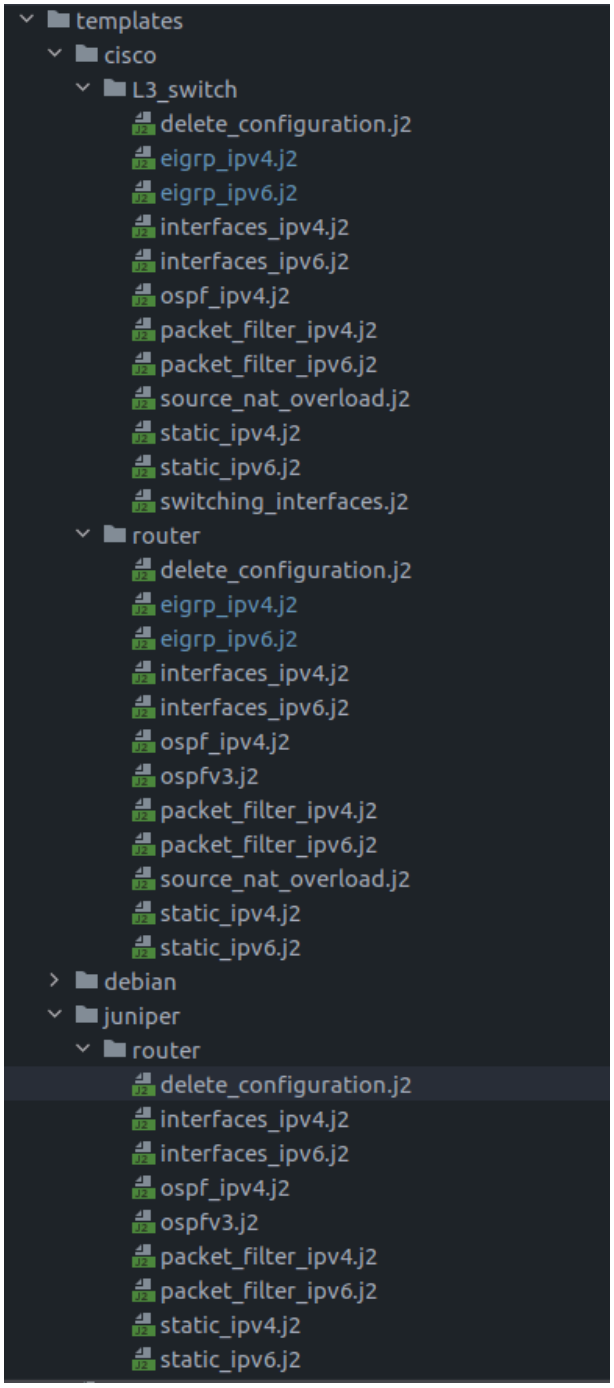
```
---
ansible_python_interpreter: ~/.local/share/virtualenvs/
    network_automation_ansible-jyRQ_HBH/bin/python
ansible_network_os: junos
ansible_connection: network_cli
ansible_user: juniper
ansible_password: "{{ vault_ansible_password }}"
napalm_provider:
  hostname: "{{ ansible_host }}"
  username: "{{ ansible_user }}"
  dev_os: "{{ ansible_network_os }}"
  password: "{{ ansible_password }}"
```

Výpis 28: Obsah souboru vars skupiny juniper

```
---
ansible_user: gns3
ansible_password: "{{ vault_ansible_password }}"
ansible_become_pass: "{{ vault_ansible_become_pass }}"

influxdb_ansible_database_name: "monitoring_ansible"
influxdb_nornir_database_name: "monitoring_nornir"
influxdb_telegraf_database_name: "monitoring_telegraf"
influxdb_admin_username: monitoring
influxdb_admin_password: "{{ vault_influxdb_admin_password }}"
```

Výpis 29: Obsah souboru vars skupiny linux

G Příklady Jinja2 šablon - Ansible projekt



Obrázek 43: Struktura adresáře **templates**

```
{% if ospfv3_config is defined %}
ipv6 unicast-routing
!
ipv6 router ospf {{ ospfv3_config.process }}
 router-id {{ ospfv3_config.router_id }}
{% if ospfv3_config.passive_interfaces is defined %}
{% for interface in ospfv3_config.passive_interfaces %}
 passive-interface {{ interface }}
{% endfor %}
{% endif %}
{% if ospfv3_config.redistribute is defined %}
{% if ospfv3_config.redistribute.static %}
 redistribute static
{% endif %}
{% if ospfv3_config.redistribute.eigrp is defined %}
{% for AS_number in ospfv3_config.redistribute.eigrp.AS_numbers %}
 redistribute eigrp {{ AS_number }}
{% endfor %}
{% endif %}
{% if ospfv3_config.redistribute.default %}
 default-information originate always
{% endif %}
{% endif %}
!
{% for int in ospfv3_config.interfaces %}
interface {{ int.name }}
 ipv6 ospf {{ ospfv3_config.process }} area {{ int.area_number }}
!
{% endfor %}
end
{% endif %}
```

Výpis 30: Obsah souboru ospfv3.j2 pro Cisco router

```
{% if ospfv3_config is defined %}
set routing-options router-id {{ ospfv3_config.router_id }}
{% for int, int_dict in ospfv3_config.interfaces.items() | sort %}
{% if int_dict.passive %}
set protocols ospf3 area {{ int_dict.area_number }} interface {{ int }} passive
```

```
{% else %}
set protocols ospf3 area {{ int_dict.area_number }} interface {{ int }}
{% endif %}
{% endfor %}
{% if ospfv3_config.redistribute_static is defined %}
set policy-options policy-statement {{ ospfv3_config.redistribute_static.
    policy_name }} term {{ ospfv3_config.redistribute_static.term_name }} from
    protocol static
set policy-options policy-statement {{ ospfv3_config.redistribute_static.
    policy_name }} term {{ ospfv3_config.redistribute_static.term_name }} then
    accept
set protocols ospf3 export {{ ospfv3_config.redistribute_static.policy_name }}
{% endif %}
{% endif %}
```

Výpis 31: Obsah souboru ospfv3.j2 pro Juniper router

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>Packet counters export</title>
   <meta charset="utf-8">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <style>
      p {
      font-weight: bold;
      font-size: 1.2rem;
      }

      h1 { text-align: center; }

      table {
      font-family: Arial, Helvetica, sans-serif;
      border-collapse: collapse;
      width: 100%;
      }

      td, th {
      border: 1px solid #ddd;
      padding: 8px;
```

```
      text-align: center;
      }

      tr:nth-child(even){background-color: #f2f2f2;}

      tr:hover {background-color: #ddd;}

      th {
      background-color: #2e81a1;
      color: white;
      }
   </style>
</head>
<body>
<h1>Packet counters tables</h1>
<p>Devices:</p>
<ul>
{% set counter = {'host_id': 0} %}
{% macro increment(dct, key, inc=1)%}
   {% if dct.update({key: dct[key] + inc}) %} {% endif %}
{% endmacro %}
{% for host in all_hosts %}
{% set id = "#" ~ host %}
   <li><a href={{ id }}>{{ host }}</a></li>
{% endfor %}
</ul>
{% for host_data in aggregated_dict %}
   <h2 id={{ all_hosts[counter.host_id] | string }}>{{ all_hosts[counter.
       host_id] | string }}</h2>
<table>
<thead>
   <tr>
      <th>interface</th>
{% for header in sorted_header %}
      <th>{{ header }}</th>
{% endfor %}
</tr>
</thead>
<tbody>
```

```
{% for int_name, int_dict in host_data.items() | sort %}
<tr>
   <td>{{ int_name }}</td>
{% for packet_counter in sorted_header %}
   <td>{{ int_dict[packet_counter] }}</td>
{% endfor %}
</tr>
{% endfor %}
</tbody>
</table>
{{ increment(counter, 'host_id') }}
{% endfor %}
</body>
</html>
```

Výpis 32: Obsah souboru packets_counter.j2

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>Devices facts export</title>
   <meta charset="utf-8">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <style>
      h1 { text-align: center; }

      table {
      font-family: Arial, Helvetica, sans-serif;
      border-collapse: collapse;
      width: 100%;
      }

      td, th {
      border: 1px solid #ddd;
      padding: 8px;
      text-align: center;
      }

      tr:nth-child(even){background-color: #f2f2f2;}
```

```
      tr:hover {background-color: #ddd;}

      th {
      background-color: #2e81a1;
      color: white;
      }
   </style>
</head>
<body>
<h1>Devices facts export</h1>
<table>
<thead>
   <tr>
{% for header in sorted_header %}
      <th>{{ header }}</th>
{% endfor %}
   </tr>
</thead>
<tbody>
{% for host_data in aggregated_dict %}
<tr>
{% for sorted_key in sorted_header %}
{% set lowercase_key = sorted_key | lower %}
{% set value = host_data[lowercase_key] %}
{% if value == "None" or "" %}
{% set value = "-" %}
{% endif %}
   <td>{{ value }}</td>
{% endfor %}
</tr>
{% endfor %}
</tbody>
</table>
</body>
</html>
```

Výpis 33: Obsah souboru facts.j2

```
{% if vsftpd_config is defined %}
listen=NO
```

```
listen_ipv6=YES
anonymous_enable=NO
local_enable=YES
write_enable=YES
local_umask=022
dirmessage_enable=YES
use_localtime=YES
xferlog_enable=YES
connect_from_port_20=YES
chroot_local_user=YES
secure_chroot_dir=/var/run/vsftpd/empty
pam_service_name=vsftpd
pasv_enable=Yes
pasv_min_port=10000
pasv_max_port=11000
user_sub_token=$USER
local_root=/home/$USER/ftp
userlist_enable=YES
userlist_file=/etc/vsftpduserlist.conf
userlist_deny=NO
{% if vsftpd_config.ssl is defined and vsftpd_config.ssl.enabled %}
ssl_enable=YES
rsa_cert_file={{ vsftpd_config.ssl.cert_file }}
rsa_private_key_file={{ vsftpd_config.ssl.private_key_file }}
allow_anon_ssl=NO
force_local_data_ssl=YES
force_local_logins_ssl=YES
ssl_tlsv1=YES
ssl_sslv2=NO
ssl_sslv3=NO
require_ssl_reuse=NO
ssl_ciphers=HIGH
{% endif %}
{% endif %}
```

Výpis 34: Obsah souboru vsftpd.j2 pro konfiguraci FTP(S) serveru

```
[global_tags]

[agent]
```

```
  interval = "20s"
  debug = false
  hostname = "{{ telegraf_ip_address }}"

###############################################################################
#                                  OUTPUTS                                    #
###############################################################################

[[outputs.influxdb]]
  urls = ["{{ influx_db_url }}"]
  database = "{{ influxdb_telegraf_database_name }}"
  username = "{{ influxdb_admin_username }}"
  password = "{{ influxdb_admin_password }}"

###############################################################################
#                                  INPUTS                                     #
###############################################################################

[[inputs.cpu]]
percpu = true
totalcpu = true
collect_cpu_time = false

[[inputs.system]]
```

Výpis 35: Obsah souboru telegraf.j2 pro konfiguraci Telegraf agenta

H Playbooky - Ansible projekt

```
- name: OSPFv2 configuration
  hosts: routers
  connection: local
  gather_facts: no
  tags: ipv4
  vars:
   templates_folder: 'templates/{{ vendor }}/{{ dev_type }}'
   staging_ospf_folder: 'files/staging/ospf'
   parsed_ospf_conf_file: '{{ staging_ospf_folder }}/{{ inventory_hostname }}.
       conf'
   commit_config: true
  tasks:
   - block:
      - name: Create folders for parsed ospf configuration
        file:
         path: "{{ staging_ospf_folder }}"
         state: directory
        check_mode: no

      - name: Set no conf commit variable (enable testing mode)
        set_fact:
         commit_config: false
        when: ansible_check_mode

      - name: Parse ospf_ipv4 template
        template:
         src: '{{ templates_folder }}/ospf_ipv4.j2'
         dest: '{{ parsed_ospf_conf_file }}'
        check_mode: no

      - name: Load and commit configuration
        napalm_install_config:
         provider: "{{ napalm_provider }}"
         config_file: '{{ parsed_ospf_conf_file }}'
         replace_config: false
         commit_changes: "{{ commit_config }}"
        register: result
```

```
    - name: Clean temporary folders/files
      file:
       state: absent
       path: "{{ staging_ospf_folder }}"
      check_mode: no

    - name: Print differences in configs
      debug:
       msg: "{{ result | to_nice_yaml }}"
    when: ospf_config is defined

- name: OSPFv3 configuration
  hosts: routers
  connection: local
  gather_facts: no
  tags: ipv6
  vars:
   templates_folder: 'templates/{{ vendor }}/{{ dev_type }}'
   staging_ospf_folder: 'files/staging/ospf'
   parsed_ospf_conf_file: '{{ staging_ospf_folder }}/{{ inventory_hostname }}.
       conf'
   commit_config: true
  tasks:
   - block:
     - name: Create folders for parsed ospf configuration
       file:
        path: "{{ staging_ospf_folder }}"
        state: directory
       check_mode: no

     - name: Set no conf commit variable (enable testing mode)
       set_fact:
        commit_config: false
       when: ansible_check_mode

     - name: Parse ospfv3 template
       template:
        src: '{{ templates_folder }}/ospfv3.j2'
```

```
        dest: '{{ parsed_ospf_conf_file }}'
      check_mode: no

    - name: Load and commit configuration
      napalm_install_config:
        provider: "{{ napalm_provider }}"
        config_file: '{{ parsed_ospf_conf_file }}'
        replace_config: false
        commit_changes: "{{ commit_config }}"
      register: result

    - name: Clean temporary folders/files
      file:
        state: absent
        path: "{{ staging_ospf_folder }}"
      check_mode: no

    - name: Print differences in configs
      debug:
        msg: "{{ result | to_nice_yaml }}"
    when: ospfv3_config is defined
```

Výpis 36: Obsah playbooku ospf_configuration.yml

```
---
- name: Export device configuration
  hosts: network_devices
  connection: local
  gather_facts: no
  vars:
   export_running_conf_folder: 'files/export/running_configuration'
  tags: run_conf
  tasks:
   - block:
     - name: Get device running configuration
       napalm_get_facts:
         provider: "{{ napalm_provider }}"
         filter: config
       register: result
```

```
    - name: Create folders for exports
      file:
       path: "{{ export_running_conf_folder }}"
      state: directory

    - name: Write content to export folder
      copy:
       content: "{{ result['ansible_facts']['napalm_config']['running'] }}"
       dest: '{{ export_running_conf_folder }}/{{ inventory_hostname }}.conf'

- name: Export IPv4 routes
  hosts: routers, cisco_l3_switches
  connection: local
  gather_facts: no
  vars:
   export_ip_routes_folder: 'files/export/ip_routes'
    commands:
      juniper: [ show route ]
      cisco: [ show ip route ]
   ipv6_pattern: "inet6.0:"
   ipv4_pattern: "inet.0"
  tags:
   - ipv4_routes
   - ip_routes
  tasks:
   - block:
      - name: Get IPv4 routes
        napalm_cli:
         provider: "{{ napalm_provider }}"
         args:
            commands: "{{ commands[vendor] }}"
        register: result

      - name: Access result of command
        set_fact:
         result: "{{ result['cli_results'][commands[vendor][0]] | trim }}"

      - name: Set no IPv4 routes parameter - juniper
        set_fact:
```

```
        result: ""
      when: vendor == 'juniper' and ipv4_pattern not in result

    - name: Parse juniper IPv4 routes
      set_fact:
       result: "{{ result.split(ipv6_pattern)[0] }}"
      when: vendor == 'juniper' and ipv6_pattern in result

    - name: Create folders for exports
      file:
       path: "{{ export_ip_routes_folder }}"
       state: directory

    - name: Write content to export folder
      copy:
       content: "{{ result }}"
       dest: '{{ export_ip_routes_folder }}/{{ inventory_hostname }}_ipv4.txt'
      when: result != ""
    when: vendor in commands


- name: Export IPv6 routes
  hosts: routers, cisco_l3_switches
  connection: local
  gather_facts: no
  vars:
   export_ip_routes_folder: 'files/export/ip_routes'
   commands:
      juniper: [ show route ]
      cisco: [ show ipv6 route ]
  ipv6_pattern: "inet6.0:"
  ipv4_pattern: "inet.0"
  tags:
   - ipv6_routes
   - ip_routes
  tasks:
   - block:
      - name: Get IPv6 routes
        napalm_cli:
```

```
      provider: "{{ napalm_provider }}"
      args:
        commands: "{{ commands[vendor] }}"
    register: result

  - name: Access result of command
    set_fact:
      result: "{{ result['cli_results'][commands[vendor][0]] | trim }}"

  - name: Set no IPv6 routes parameter - juniper
    set_fact:
      result: ""
    when: vendor == 'juniper' and ipv6_pattern not in result

  - name: Parse juniper IPv6 routes
    set_fact:
      result: "{{ ipv6_pattern + result.split(ipv6_pattern)[1] }}"
    when: vendor == 'juniper' and ipv6_pattern in result

  - name: Create folders for exports
    file:
      path: "{{ export_ip_routes_folder }}"
      state: directory

  - name: Write content to export folder
    copy:
      content: "{{ result }}"
      dest: '{{ export_ip_routes_folder }}/{{ inventory_hostname }}_ipv6.txt'
    when: result != ""
  when: vendor in commands

- name: Export packet filters info
  hosts: routers, cisco_l3_switches
  connection: local
  gather_facts: no
  vars:
    export_packet_filter_folder: 'files/export/packet_filter'
    commands:
      juniper: [show configuration firewall]
```

```
      cisco: [show access-lists]
  tags:
   - packet_filters
   - acl
  tasks:
   - block:
      - name: Get packet filters info
        napalm_cli:
         provider: "{{ napalm_provider }}"
         args:
            commands: "{{ commands[vendor] }}"
        register: result

      - name: Access result of command
        set_fact:
         result: "{{ result['cli_results'][commands[vendor][0]] | trim }}"

      - name: Create folders for exports
        file:
         path: "{{ export_packet_filter_folder }}"
        state: directory

      - name: Write content to export folder
        copy:
         content: "{{ result }}"
         dest: '{{ export_packet_filter_folder }}/{{ inventory_hostname }}.txt'
        when: result != ""
      when: vendor in commands

- name: Export packet counters data to HTML
  hosts: network_devices
  connection: local
  gather_facts: no
  tags: packets_counter
  vars:
   templates_folder: 'templates'
   export_folder: 'files/export'
   sorted_header: [ "rx_broadcast_packets", "rx_discards", "rx_errors",
   "rx_multicast_packets", "rx_octets", "rx_unicast_packets",
```

```
    "tx_discards", "tx_errors", "tx_octets", "tx_unicast_packets" ]
  tasks:
   - block:
      - name: Get interfaces packet counters
        napalm_get_facts:
         provider: "{{ napalm_provider }}"
         filter: interfaces_counters
        register: result

      - name: Access result of command
        set_fact:
         result: "{{ result['ansible_facts']['napalm_interfaces_counters'] }}"
         all_hosts: "{{ play_hosts }}"

      - name: Aggregate all results
        set_fact:
         aggregated_dict: "{{ ansible_play_hosts | map('extract', hostvars, '
             result') }}"
        run_once: yes

      - name: Create folders for exports
        file:
         path: "{{ export_folder }}"
         state: directory

      - name: Parse packets_counter template
        template:
         src: '{{ templates_folder }}/packets_counter.j2'
         dest: '{{ export_folder }}/packets_counter.html'
        run_once: yes

- name: Export devices facts to HTML
  hosts: network_devices
  connection: local
  gather_facts: no
  tags: facts
  vars:
   templates_folder: 'templates'
   export_folder: 'files/export'
```

```
 sorted_header: ["hostname", "FQDN", "vendor", "model", "serial_number", "
     os_version", "uptime"]
tasks:
 - block:
    - name: Get devices facts
      napalm_get_facts:
       provider: "{{ napalm_provider }}"
       filter: facts
      register: result

    - name: Access result of command
      set_fact:
       result: "{{ result['ansible_facts']['napalm_facts'] }}"
       all_hosts: "{{ play_hosts }}"

    - name: Insert vendor value (only Olive images)
      set_fact:
       result: "{{ result | default({}) | combine ({ 'vendor' : vendor | title
            }) }}"
      when: image == "olive"

    - name: Parse os_version (only Cisco devices)
      set_fact:
       result: "{{ result | default({}) | combine ({ 'os_version' : result['
           os_version'].split(', ')[1] }) }}"
      when: vendor == "cisco"

    - name: Parse uptime
      set_fact:
       result: "{{ result | default({}) | combine ({ 'uptime' : '%H:%M:%S' |
           strftime(result['uptime'] - 3600 ) }) }}"

    - name: Aggregate all results
      set_fact:
       aggregated_dict: "{{ ansible_play_hosts | map('extract', hostvars, '
           result') }}"
      run_once: yes

    - name: Create folders for exports
```

```
      file:
       path: "{{ export_folder }}"
       state: directory

    - name: Parse facts template
      template:
       src: '{{ templates_folder }}/facts.j2'
       dest: '{{ export_folder }}/facts.html'
      run_once: yes
```

Výpis 37: Obsah playbooku network_info_exporter.yml

```
---
- name: Backup network device configuration
  hosts: network_devices
  connection: local
  gather_facts: no
  vars:
   backups_folder: 'files/backups/{{inventory_hostname}}'
  tasks:
   - block:
     - name: Get timestamp from local system
       shell: "date +%Y-%m-%d%H-%M-%S"
       register: timestamp

     - name: Get device running configuration
       napalm_get_facts:
        provider: "{{ napalm_provider }}"
        filter: config
       register: result

     - name: Create folders for backups
       file:
        path: "{{ backups_folder }}"
        state: directory

     - name: Write content to backups folder
       copy:
        content: "{{ result['ansible_facts']['napalm_config']['running'] }}"
```

```
        dest: '{{ backups_folder }}/{{ inventory_hostname }}_{{ timestamp.
            stdout[0:10] }}.conf'
```

Výpis 38: Obsah playbooku backup_configuration.yml

```
---
- name: Deletion of network devices configuration
  hosts: network_devices
  connection: local
  gather_facts: no
  vars:
   templates_folder: 'templates/{{ vendor }}/{{ dev_type }}'
   staging_delete_folder: 'files/staging/delete'
   parsed_delete_commands_files: '{{ staging_delete_folder }}/{{
       inventory_hostname }}.txt'
   commit_config: true
  tasks:
   - block:
      - name: Create folders for parsed delete commands
        file:
         path: "{{ staging_delete_folder }}"
         state: directory
        check_mode: no

      - name: Set no conf commit variable (enable testing mode)
        set_fact:
         commit_config: false
        when: ansible_check_mode

      - name: Parse delete_configuration template
        template:
         src: '{{ templates_folder }}/delete_configuration.j2'
         dest: '{{ parsed_delete_commands_files }}'
        check_mode: no

      - name: Load and commit configuration
        napalm_install_config:
         provider: "{{ napalm_provider }}"
         config_file: '{{ parsed_delete_commands_files }}'
         replace_config: false
```

```
        commit_changes: "{{ commit_config }}"
      register: result

    - name: Clean temporary folders/files
      file:
       state: absent
       path: "{{ staging_delete_folder }}"
      check_mode: no

    - name: Print differences in configs
      debug:
       msg: "{{ result | to_nice_yaml }}"
    when: delete_config is defined
```

Výpis 39: Obsah playbooku delete_configuration.yml

```
---
- name: Restore network device configuration
  hosts: network_devices
  connection: local
  gather_facts: no
  vars:
   backups_folder: 'files/backups/{{inventory_hostname}}'
   commit_config: true
  tasks:
   - block:
     - name: Set no conf commit variable (enable testing mode)
       set_fact:
        commit_config: false
       when: ansible_check_mode

     - name: Load and commit configuration
       napalm_install_config:
        provider: "{{ napalm_provider }}"
        config_file: '{{ backups_folder }}/{{ inventory_hostname }}_{{
            restore_config.running_config_date }}.conf'
        replace_config: true
        commit_changes: "{{ commit_config }}"
       register: result
```

```
    - name: Print differences in configs
      debug:
       msg: "{{ result | to_nice_yaml }}"
    when: restore_config is defined and restore_config.running_config_date is
        defined
```

Výpis 40: Obsah playbooku restore_configuration.yml

```
---
- name: Setup and Configure FTPS server
  hosts: ubuntu_servers
  gather_facts: no
  become: yes
  tags: vsftpd
  vars:
   templates_folder: 'templates/{{ vendor }}/{{ dev_type }}'
   vsftpd_conf_path: '/etc/vsftpd.conf'
   vsftpd_userlist_path: '/etc/vsftpduserlist.conf'
   certs_folder_path: '/etc/certs'
  tasks:
   - name: Ensure FTP group exists
      tags: user
     group:
      name: "{{ vsftpd_config.vsftpd_user.name }}"
      state: present

   - name: Create FTP user
     tags: user
     user:
      name: "{{ vsftpd_config.vsftpd_user.name }}"
      state: present
      group: "{{ vsftpd_config.vsftpd_user.name }}"
      groups: "{{ vsftpd_config.vsftpd_user.groups }}"
      shell: /bin/bash
      home: "{{ vsftpd_config.vsftpd_user.home }}"
      password: "{{ vsftpd_config.vsftpd_user.password }}"
      update_password: on_create

   - name: Install VSFTPD
     apt:
```

```
    pkg: vsftpd
    state: present

- name: Enable VSFTPD during boot
  service:
    name: vsftpd
    state: started
    enabled: yes

 - block:
    - name: Create certs directory
      file:
      path: "{{ certs_folder_path }}"
      state: directory
      owner: root
      group: root
      mode: '0755'

 - name: Generate an OpenSSL private key.
    openssl_privatekey:
     path: "{{ vsftpd_config.ssl.private_key_file }}"

 - name: Generate an OpenSSL CSR.
    openssl_csr:
    path: "{{ vsftpd_config.ssl.openssl_csr }}"
    privatekey_path: "{{ vsftpd_config.ssl.private_key_file }}"
    common_name: "{{ vsftpd_config.ssl.cn }}"

- name: Generate a Self Signed OpenSSL certificate.
  openssl_certificate:
    path: "{{ vsftpd_config.ssl.cert_file }}"
    privatekey_path: "{{ vsftpd_config.ssl.private_key_file }}"
    csr_path: "{{ vsftpd_config.ssl.openssl_csr }}"
    provider: selfsigned
when: vsftpd_config.ssl is defined and vsftpd_config.ssl.enabled

- name: Create base FTP directory
  file:
    path: '/home/{{ vsftpd_config.vsftpd_user.name }}/ftp'
```

```
    state: directory
    owner: nobody
    group: nogroup
    mode: '0555'

- name: Create test FTP directory (for ftp users)
  file:
    path: '/home/{{ vsftpd_config.vsftpd_user.name }}/ftp/test'
    state: directory
    owner: '{{ vsftpd_config.vsftpd_user.name }}'
    group: '{{ vsftpd_config.vsftpd_user.name }}'
    mode: '0755'

- name: Parse vsftpd template
  template:
    src: '{{ templates_folder }}/vsftpd.j2'
    dest: '{{ vsftpd_conf_path }}'
    owner: root
    group: root
    mode: 0644
    force: yes
  notify:
   - restart vsftpd

- name: Create VSFTPD userlist
  copy:
    dest: '{{ vsftpd_userlist_path }}'
    owner: root
    group: root
    mode: 0644
    force: yes
    content: |
    {{ vsftpd_config.vsftpd_user.name }}
  notify:
   - restart vsftpd

handlers:
  - name: restart vsftpd
    service:
```

```
      name: vsftpd
      state: restarted

- name: Setup TIG Stack
  hosts: Server1
  gather_facts: yes
  become: yes
  tags: monitoring
  vars:
   influx_gpg_key_url: 'https://repos.influxdata.com/influxdb.key'
   influx_repository: 'deb https://repos.influxdata.com/ubuntu bionic stable'
   grafana_gpg_key_url: 'https://packages.grafana.com/gpg.key'
   grafana_repository: 'deb https://packages.grafana.com/oss/deb stable main'
   templates_folder: 'templates/{{ vendor }}/{{ dev_type }}'
   telegraf_conf_path: '/etc/telegraf/telegraf.conf'
  tasks:
   - name: Import InfluxDB GPG signing key
     apt_key:
      url: "{{ influx_gpg_key_url }}"
      state: present

   - name: Add InfluxDB repository
     apt_repository:
      repo: "{{ influx_repository }}"
      state: present

   - name: Import Grafana GPG signing key
     apt_key:
      url: "{{ grafana_gpg_key_url }}"
      state: present

   - name: Add Grafana repository
     apt_repository:
      repo: "{{ grafana_repository }}"
      state: present

   - name: Install InfluxDB
     apt:
      pkg: influxdb
```

```
    state: present

- name: Enable InfluxDB during boot
  service:
    name: influxdb
    state: started
    enabled: yes

- name: Install pip3
  tags: db
    apt:
    name: python3-pip
    state: present

- name: Install influxdb python package
  pip:
    name: influxdb
    state: present

- name: Create an admin DB user
  community.general.influxdb_user:
    user_name: "{{ influxdb_admin_username }}"
    user_password: "{{ influxdb_admin_password }}"
    admin: yes
    hostname: "{{ influxdb_ip_address }}"
    state: present

- name: Create databases for Nornir and Ansible project
  community.general.influxdb_database:
    hostname: "{{ influxdb_ip_address }}"
    database_name: "{{ item }}"
    state: present
  with_items:
    - "{{ influxdb_ansible_database_name }}"
    - "{{ influxdb_nornir_database_name }}"
    - "{{ influxdb_telegraf_database_name }}"

- name: Install Grafana
  apt:
```

```
    pkg: grafana
    state: present

- name: Enable Grafana during boot
  service:
    name: grafana-server
    state: started
    enabled: yes

- name: Install telegraf
  apt:
    name: telegraf
    state: latest

- name: Parse telegraf template
  template:
    src: '{{ templates_folder }}/telegraf.j2'
    dest: '{{ telegraf_conf_path }}'
    owner: root
    group: root
    mode: 0644
    force: yes
  notify:
    - restart telegraf

- name: Enable telegraf agent during boot
  service:
    name: telegraf
    state: started
    enabled: yes

- name: Create influxdb datasources
  community.grafana.grafana_datasource:
    name: "{{ item.name }}"
    grafana_url: "{{ grafana_url }}"
    ds_type: "{{ item.ds_type }}"
    ds_url: "{{ item.url }}"
    database: "{{ item.database }}"
    user: "{{ item.user }}"
```

```
      password: "{{ item.password }}"
      state: present
    with_items: "{{ grafana_data_sources }}"

  handlers:
   - name: restart telegraf
     service:
      name: telegraf
      state: restarted
```

Výpis 41: Obsah playbooku linux_configuration.yml

```
---
- name: Get and Write NAPALM data to DB
  hosts: network_devices
  gather_facts: no
  vars:
   used_influxdb_ip_address: 10.10.10.6
   used_influxdb_ansible_database_name: "monitoring_ansible"
  tasks:
   - name: Get env_details and device facts
     napalm_get_facts:
      provider: "{{ napalm_provider }}"
      filter: environment,facts
     register: result

   - name: Get current UTC time
     command: "date '+%Y-%m-%d %H:%M:%S'"
     delegate_to: localhost
     register: date_utc
     environment:
      TZ: UTC
     run_once: yes

   - name: Access HW details (except Olive)
     set_fact:
      cpu_usage: "{{ (result['ansible_facts']['napalm_environment']['cpu
          ']['0']['%usage']) }}"
     when: image != "olive"
```

```
- name: Access facts (uptime)
  set_fact:
   uptime: "{{ result['ansible_facts']['napalm_facts']['uptime'] }}"

- name: Write HW details into DB
  community.general.influxdb_write:
   hostname: "{{ used_influxdb_ip_address }}"
   database_name: "{{ used_influxdb_ansible_database_name }}"
   data_points:
      - measurement: hw_details
        tags:
         host: "{{ inventory_hostname }}"
        time: "{{ date_utc.stdout }}"
        fields:
         cpu_usage: "{{ cpu_usage }}"
  throttle: 1
  when: cpu_usage is defined

- name: Write device facts into DB
  community.general.influxdb_write:
   hostname: "{{ used_influxdb_ip_address }}"
   database_name: "{{ used_influxdb_ansible_database_name }}"
   data_points:
      - measurement: device_facts
        tags:
         host: "{{ inventory_hostname }}"
        time: "{{ date_utc.stdout }}"
        fields:
         uptime: "{{ uptime }}"
 throttle: 1
 when: uptime is defined
```

Výpis 42: Obsah playbooku db_handler.yml

I Výpisy do konzole - Ansible projekt

```
TASK [Print differences in configs] ********
ok: [R1] =>
  msg: |-
    changed: true
    failed: false
    msg: |-
        +ipv6 router ospf 1
        + router-id 1.1.1.1
        + passive-interface FastEthernet0/1
        +interface FastEthernet0/0
        + ipv6 ospf 1 area 0
        +interface FastEthernet0/1
        + ipv6 ospf 1 area 0
ok: [R2] =>
  msg: |-
    changed: true
    failed: false
    msg: |-
        [edit protocols]
        +   ospf3 {
        +       area 0.0.0.0 {
        +           interface em1.0;
        +           interface em2.0;
        +       }
        +   }
ok: [R3] =>
  msg: |-
    changed: true
    failed: false
    msg: |-
        +ipv6 router ospf 1
        + router-id 3.3.3.3
        + redistribute eigrp 1
        +interface FastEthernet0/0
        + ipv6 ospf 1 area 0
        +interface FastEthernet0/1
        + ipv6 ospf 1 area 0
```

(a) Po prvním provedení Ansible hry

```
TASK [Print differences in configs] ******
*
ok: [R1] =>
  msg: |-
    changed: false
    failed: false
    msg: ''
ok: [R2] =>
  msg: |-
    changed: false
    failed: false
    msg: ''
ok: [R3] =>
  msg: |-
    changed: false
    failed: false
    msg: ''
```

(b) Po druhém provedení Ansible hry

Obrázek 44: Výpis rozdílů v konfiguracích po provedení Ansible hry **OSPFv3 configuration**

Obrázek 45: Výpis konfigurovaných VLAN - zařízení MLS1

J Sběr dat ze síťových zařízení - Ansible projekt

```
- name: Show configured VLANs
  hosts: cisco_l3_switches
  connection: local
  gather_facts: no
  tags:
   - vlan
  tasks:
   - block:
      - name: Get configured VLANs
        napalm_get_facts:
         provider: "{{ napalm_provider }}"
         filter: vlans
        register: result

      - name: Print configured VLANs
        debug:
         msg: "{{ result | to_nice_yaml }}"
```

Výpis 43: Struktura Ansible hry **Show configured VLANs**

```
- name: Show OSPFv3 neighbors
  hosts: routers
  connection: local
  gather_facts: no
  tags:
   - ospfv3
  vars:
   commands:
      juniper: [show ospf3 neighbor]
      cisco: [show ipv6 ospf neighbor]
  tasks:
   - block:
      - name: Get OSPFv3 neighbors
        napalm_cli:
         provider: "{{ napalm_provider }}"
         args:
            commands: "{{ commands[vendor] }}"
        register: result
```

```
- name: Print OSPFv3 neighbors
  debug:
   msg: "{{ result | to_nice_yaml }}"
when: vendor in commands
```

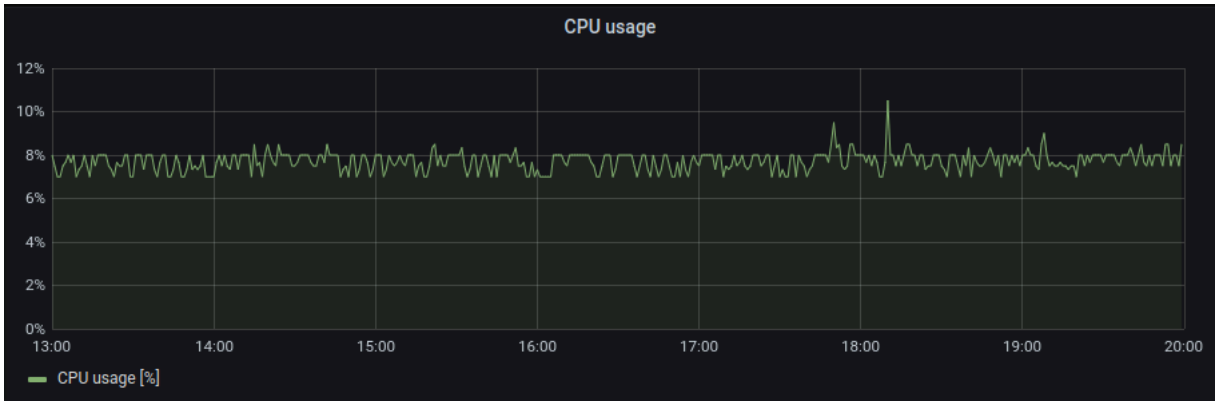
Výpis 44: Struktura Ansible hry **Show OSPFv3 neighbors**

K Grafy - Ansible + TIG stack

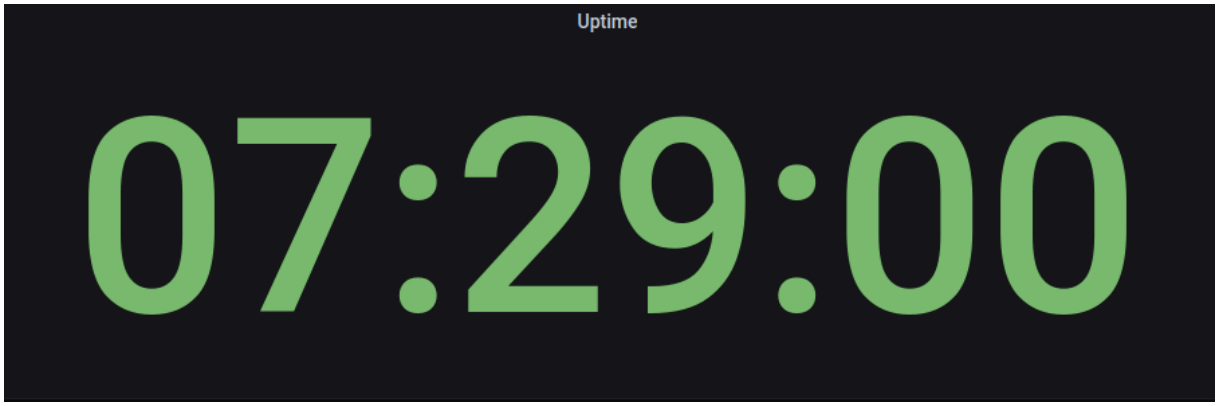
K.1 Síťové zařízení R2

Obrázek 46: Uptime síťového zařízení R2

K.2 Síťové zařízení R3

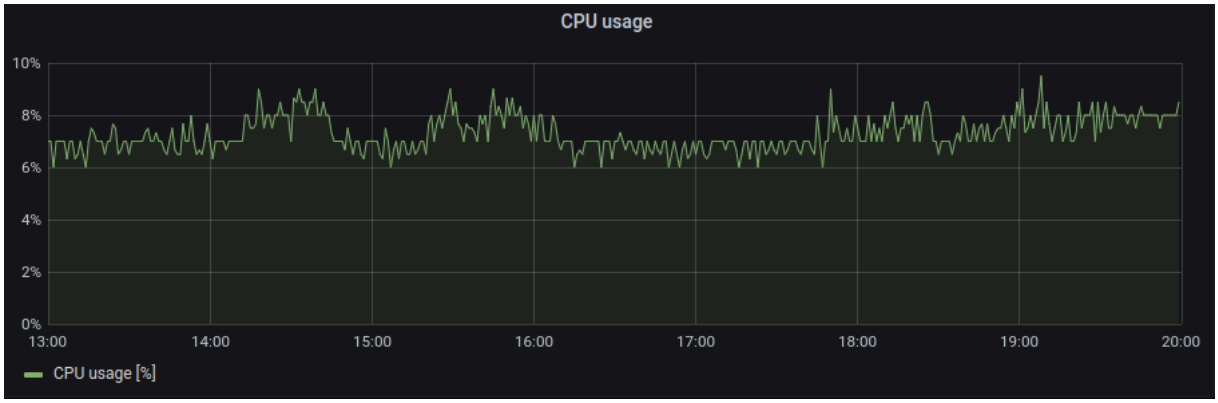


Obrázek 47: Vytížení procesoru síťového zařízení R3



Obrázek 48: Uptime síťového zařízení R3

K.3 Síťové zařízení MLS1



Obrázek 49: Vytížení procesoru síťového zařízení MLS1

Obrázek 50: Uptime síťového zařízení MLS1

L Inventář - Nornir projekt

```
---
R1:
  hostname: 10.10.10.2
  groups:
    - cisco
  data:
    type: network_device
    vendor: cisco
    dev_type: router
    image: c7200
R2:
  hostname: 10.10.10.3
  groups:
    - juniper
  data:
    type: network_device
    vendor: juniper
    dev_type: router
    image: olive
R3:
  hostname: 10.10.10.4
  groups:
    - cisco
  data:
    type: network_device
    vendor: cisco
    dev_type: router
    image: c7200
MLS1:
  hostname: 10.10.10.5
  groups:
    - cisco
  data:
    type: network_device
    vendor: cisco
    dev_type: L3_switch
    image: IOSvL2
```

```
Server1:
   hostname: 10.10.10.6
   groups:
      - linux
   data:
      type: network_device
      vendor: debian
      dev_type: ubuntu_server
```

Výpis 45: Obsah souboru hosts.yml

```
---
cisco:
   platform: "ios"
   connection_options:
      napalm:
         extras:
            optional_args:
               global_delay_factor: 1
      netmiko:
         extras:
            global_delay_factor: 1

juniper:
   platform: "junos"

linux:
   platform: "linux" # pro netmiko driver
   connection_options:
      netmiko:
         extras:
            global_delay_factor: 3
```

Výpis 46: Obsah souboru group.yml

```
---
interfaces_ipv4:
   FastEthernet0/0:
      description: connected to R2 on port em1
      net: 192.168.2.1
      mask: 255.255.255.0
```

```
      routed_physical_port: true
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to SW1 on port e0
      net: 192.168.1.1
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
      speed: auto
ospf_config:
   process: 1
   router_id: 1.1.1.1
   passive_interfaces: [ FastEthernet0/1 ]
   networks:
      - ip: 192.168.1.0
        wildcard: 0.0.0.255
        area_number: 0
      - ip: 192.168.2.0
        wildcard: 0.0.0.255
        area_number: 0
interfaces_ipv6:
   FastEthernet0/0:
      description: connected to R2 on port em1
      networks:
         - net: "2001:db8:1001:2::1"
           prefix_length: /64
           eui_format_auto: false
      routed_physical_port: true
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to SW1 on port e0
      networks:
         - net: "2001:db8:1001:1::1"
           prefix_length: /64
           eui_format_auto: false
      routed_physical_port: true
      duplex: full
```

```
      speed: auto
ospfv3_config:
   process: 1
   router_id: 1.1.1.1
   passive_interfaces: [ FastEthernet0/1 ]
   interfaces:
      - name: FastEthernet0/0
        area_number: 0
      - name: FastEthernet0/1
        area_number: 0
restore_config:
   running_config_date: 2021-04-07
delete_config:
   interfaces: # routované porty
      physical: [FastEthernet0/0, FastEthernet0/1]
   ospf_config:
      processes: [1]
   ospfv3_config:
      processes:
         - number: 1
           interfaces:
            - name: FastEthernet0/0
              area_number: 0
            - name: FastEthernet0/1
              area_number: 0
```

Výpis 47: Obsah souboru R1.yml

```
---
interfaces_ipv4:
   FastEthernet0/0:
      description: connected to R2 on port em2
      net: 192.168.3.2
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to MLS1 on port g0/1
      net: 192.168.4.1
```

```
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
      speed: auto
ospf_config: # OSPFv2 konfigurace + redistribuce
   process: 1
   router_id: 3.3.3.3
   networks:
      - ip: 192.168.3.0
        wildcard: 0.0.0.255
        area_number: 0
   redistribute:
      static: false
      default: false
      eigrp:
         AS_numbers: [1]
eigrp_config:
   AS_number: 1
   passive_interfaces: []
   networks:
      - ip: 192.168.4.0
        wildcard: 0.0.0.255
   redistribute:
      ospf:
         processes:
            - number: 1
              bandwidth: 1000000
              delay: 1
              reliability: 255
              load: 1
              mtu: 1500
interfaces_ipv6:
   FastEthernet0/0:
      description: connected to R2 on port em2
      networks:
         - net: "2001:db8:1001:3::2"
           prefix_length: /64
           eui_format_auto: false
      routed_physical_port: true
```

```
      duplex: full
      speed: auto
   FastEthernet0/1:
      description: connected to MLS1 on port g0/1
      networks:
         - net: "2001:db8:1001:4::1"
           prefix_length: /64
           eui_format_auto: false
      routed_physical_port: true
      duplex: full
      speed: auto
ospfv3_config:
   process: 1
   router_id: 3.3.3.3
   interfaces:
      - name: FastEthernet0/0
        area_number: 0
   redistribute:
      static: false
      default: false
      eigrp:
         AS_numbers: [1]
restore_config:
   running_config_date: 2021-04-07
eigrp_ipv6_config:
   AS_number: 1
   router_id: 3.3.3.3
   passive_interfaces: []
   interfaces: [ FastEthernet0/1 ]
   redistribute:
      ospf:
         processes:
            - number: 1
              bandwidth: 1000000
              delay: 1
              reliability: 255
              load: 1
              mtu: 1500
delete_config:
```

```
   interfaces:
      physical: [FastEthernet0/0, FastEthernet0/1]
   eigrp_config:
      AS_numbers: [ 1 ]
   eigrp_ipv6_config:
      AS:
        - number: 1
          interfaces: [ FastEthernet0/1 ]
   ospf_config:
      processes: [1]
   ospfv3_config:
      processes:
         - number: 1
         interfaces:
            - name: FastEthernet0/0
              area_number: 0
            - name: FastEthernet0/1
              area_number: 0
```

Výpis 48: Obsah souboru R3.yml

```
---
vlans_config:
   - number: 30
     name: VLAN30
   - number: 40
     name: VLAN40
   - number: 50
     name: VLAN50
switching_interfaces:
   GigabitEthernet0/2:
      description: connected to PC2 on port eth0
      mode: access
      vlan: 30
   GigabitEthernet0/3:
      description: connected to PC3 on port eth0
      mode: access
      vlan: 40
   GigabitEthernet1/0:
      description: connected to Server1 on port ens4
```

```
      mode: access
      vlan: 50
interfaces_ipv4:
   GigabitEthernet0/1:
      description: connected to R3 on port f0/1
      net: 192.168.4.2
      mask: 255.255.255.0
      routed_physical_port: true
      duplex: full
   Vlan30:
      description: VLAN30 SVI
      net: 192.168.30.1
      mask: 255.255.255.0
   Vlan40:
      description: VLAN40 SVI
      net: 192.168.40.1
      mask: 255.255.255.0
   Vlan50:
      description: VLAN50 SVI
      net: 192.168.50.1
      mask: 255.255.255.0
eigrp_config:
   AS_number: 1
   passive_interfaces: []
   networks:
      - ip: 192.168.4.0
        wildcard: 0.0.0.255
      - ip: 192.168.30.0
        wildcard: 0.0.0.255
      - ip: 192.168.40.0
        wildcard: 0.0.0.255
      - ip: 192.168.50.0
        wildcard: 0.0.0.255
packet_filter_config:
   VLAN30in:
      type: extended
      rules:
         - deny ip 192.168.30.0 0.0.0.255 192.168.40.0 0.0.0.255
         - permit ip any any
```

```
      routed_interfaces:
        - name: Vlan30
          acl_direction: in
    VLAN40in:
      type: extended
      rules:
        - deny ip 192.168.40.0 0.0.0.255 192.168.30.0 0.0.0.255
        - permit ip any any
      routed_interfaces:
        - name: Vlan40
          acl_direction: in
interfaces_ipv6:
    GigabitEthernet0/1:
      description: connected to R3 on port f0/1
      networks:
        - net: "2001:db8:1001:4::2"
          prefix_length: /64
          eui_format_auto: false
      routed_physical_port: true
      duplex: full
      speed: auto
    Vlan30:
      description: VLAN30 SVI
      networks:
        - net: "2001:db8:1001:30::1"
          prefix_length: /64
          eui_format_auto: false
    Vlan40:
      description: VLAN40 SVI
      networks:
        - net: "2001:db8:1001:40::1"
          prefix_length: /64
          eui_format_auto: false
    Vlan50:
      description: VLAN50 SVI
      networks:
        - net: "2001:db8:1001:50::1"
          prefix_length: /64
          eui_format_auto: false
```

```
eigrp_ipv6_config:
   AS_number: 1
   router_id: 4.4.4.4
   routed_interfaces: [ Vlan30, Vlan40, Vlan50, GigabitEthernet0/1 ]
packet_filter_ipv6_config:
   VLAN30-in:
      rules:
         - deny ipv6 2001:db8:1001:30::/64 2001:db8:1001:40::/64
         - permit ipv6 any any
      routed_interfaces:
         - name: Vlan30
           acl_direction: in
   VLAN40-in:
      rules:
         - deny ipv6 2001:db8:1001:40::/64 2001:db8:1001:30::/64
         - permit ipv6 any any
      routed_interfaces:
         - name: Vlan40
           acl_direction: in
restore_config:
   running_config_date: 2021-04-07
delete_config:
   interfaces:
      virtual: [Vlan30,Vlan40,Vlan50]
      physical: [GigabitEthernet0/1, GigabitEthernet0/2, GigabitEthernet0/3,
          GigabitEthernet1/0]
   eigrp_config:
      AS_numbers: [1]
   packet_filter_config:
      Vlan30in:
         type: extended
         routed_interfaces:
            - name: Vlan30
              acl_direction: in
      Vlan40in:
         type: extended
         routed_interfaces:
            - name: Vlan40
              acl_direction: in
```

```
  eigrp_ipv6_config:
    AS:
      - number: 1
        routed_interfaces: [ Vlan30,Vlan40,Vlan50, GigabitEthernet0/1 ]
  packet_filter_ipv6_config:
    Vlan30-in:
      routed_interfaces:
        - name: Vlan30
          acl_direction: out
    Vlan40-in:
      routed_interfaces:
        - name: Vlan40
          acl_direction: out
  vlans_config:
    numbers: [30, 40, 50]
```

Výpis 49: Obsah souboru MLS1.yml

```
---
interfaces_ipv4:
  em1:
    description: '"connected to R1 on port f0/0"'
    units:
      - number: 0
        net: 192.168.2.2
        prefix_length: /24
  em2:
    description: '"connected to R3 on port f0/0"'
    units:
      - number: 0
        net: 192.168.3.1
        prefix_length: /24
ospf_config:
  router_id: 2.2.2.2
  interfaces:
    em1.0:
      passive: false
      area_number: 0
    em2.0:
      passive: false
```

```
        area_number: 0
interfaces_ipv6:
   em1:
      description: '"connected to R1 on port f0/0"'
      units:
         - number: 0
           net: "2001:db8:1001:2::2"
           prefix_length: /64
   em2:
      description: '"connected to R3 on port f0/0"'
      units:
         - number: 0
           net: "2001:db8:1001:3::1"
           prefix_length: /64
ospfv3_config:
   router_id: 2.2.2.2
   interfaces:
      em1.0:
         passive: false
         area_number: 0
      em2.0:
         passive: false
         area_number: 0
restore_config:
   running_config_date: 2021-04-07
delete_config:
   interfaces:
      em1:
         delete_whole_interface: true
         delete_units: []
      em2:
         delete_whole_interface: false
         delete_units: [0]
   ospf_config:
      router_id: 2.2.2.2
   ospfv3_config:
      router_id: 2.2.2.2
```

Výpis 50: Obsah souboru R2.yml

```yaml
---
commands: ["pwd", "ls"]
vsftpd_config:
   ssl:
      enabled: true
      rsa_cert_file: /etc/certs/vsftpd.pem
      rsa_private_key_file: /etc/certs/vsftpd.pem
   commands:
      - "apt-get install vsftpd -y"
      - "systemctl start vsftpd"
      - "systemctl enable vsftpd"
      - "mkdir /home/ftpuser/ftp"
      - "chown nobody:nogroup /home/ftpuser/ftp"
      - "chmod a-w /home/ftpuser/ftp"
      - "mkdir /home/ftpuser/ftp/test"
      - "chown ftpuser:ftpuser /home/ftpuser/ftp/test"
      - "rm /etc/vsftpd.conf"
      - "scp" # zvoleny prikaz, ve skriptu bude tento prikaz kontrolovan. Pokud
          nastane, tak se spusti netmiko_file_transfer task (SCP) pro
         prekopirovani vsftpd.conf
      - "echo ftpuser | sudo tee -a /etc/vsftpduserlist.conf"
      - "systemctl restart vsftpd"
```

Výpis 51: Obsah souboru Server1.yml

M Příklady Jinja2 šablon - Nornir projekt

```
{% if host.ospfv3_config is defined %}
ipv6 unicast-routing
!
ipv6 router ospf {{ host.ospfv3_config.process }}
 router-id {{ host.ospfv3_config.router_id }}
{% if host.ospfv3_config.passive_interfaces is defined %}
{% for interface in host.ospfv3_config.passive_interfaces %}
 passive-interface {{ interface }}
{% endfor %}
{% endif %}
{% if host.ospfv3_config.redistribute is defined %}
{% if host.ospfv3_config.redistribute.static %}
 redistribute static
{% endif %}
{% if host.ospfv3_config.redistribute.eigrp is defined %}
{% for AS_number in host.ospfv3_config.redistribute.eigrp.AS_numbers %}
 redistribute eigrp {{ AS_number }}
{% endfor %}
{% endif %}
{% if host.ospfv3_config.redistribute.default %}
 default-information originate always
{% endif %}
{% endif %}
!
{% for int in host.ospfv3_config.interfaces %}
interface {{ int.name }}
 ipv6 ospf {{ host.ospfv3_config.process }} area {{ int.area_number }}
!
{% endfor %}
end
{% endif %}
```

Výpis 52: Obsah souboru ospfv3.j2 pro Cisco router

```
{% if host.ospfv3_config is defined %}
set routing-options router-id {{ host.ospfv3_config.router_id }}
{% for int, int_dict in host.ospfv3_config.interfaces.items() | sort %}
{% if int_dict.passive %}
```

```
set protocols ospf3 area {{ int_dict.area_number }} interface {{ int }} passive
{% else %}
set protocols ospf3 area {{ int_dict.area_number }} interface {{ int }}
{% endif %}
{% endfor %}
{% if host.ospfv3_config.redistribute_static is defined %}
set policy-options policy-statement {{ host.ospfv3_config.redistribute_static.
    policy_name }} term {{ host.ospfv3_config.redistribute_static.term_name }}
    from protocol static
set policy-options policy-statement {{ host.ospfv3_config.redistribute_static.
    policy_name }} term {{ host.ospfv3_config.redistribute_static.term_name }}
    then accept
set protocols ospf3 export {{ host.ospfv3_config.redistribute_static.
    policy_name }}
{% endif %}
{% endif %}
```

Výpis 53: Obsah souboru ospfv3.j2 pro Juniper router

```
{% if host.vsftpd_config is defined %}
listen=NO
listen_ipv6=YES
anonymous_enable=NO
local_enable=YES
write_enable=YES
local_umask=022
dirmessage_enable=YES
use_localtime=YES
xferlog_enable=YES
connect_from_port_20=YES
chroot_local_user=YES
secure_chroot_dir=/var/run/vsftpd/empty
pam_service_name=vsftpd
pasv_enable=Yes
pasv_min_port=10000
pasv_max_port=11000
user_sub_token=$USER
local_root=/home/$USER/ftp
userlist_enable=YES
userlist_file=/etc/vsftpduserlist.conf
```

```
userlist_deny=NO
{% if host.vsftpd_config.ssl is defined and host.vsftpd_config.ssl.enabled %}
ssl_enable=YES
rsa_cert_file={{ host.vsftpd_config.ssl.rsa_cert_file }}
rsa_private_key_file={{ host.vsftpd_config.ssl.rsa_private_key_file }}
allow_anon_ssl=NO
force_local_data_ssl=YES
force_local_logins_ssl=YES
ssl_tlsv1=YES
ssl_sslv2=NO
ssl_sslv3=NO
require_ssl_reuse=NO
ssl_ciphers=HIGH
{% endif %}
{% endif %}
```

Výpis 54: Obsah souboru vsftpd.j2 pro konfiguraci FTP(S) serveru

N Python skripty - Nornir projekt

```
import datetime
import time

from colorama import Fore
from nornir import InitNornir
from nornir.core import Nornir
from nornir.core.exceptions import NornirSubTaskError, NornirExecutionError
from nornir.core.filter import F
from nornir_netmiko import netmiko_send_command
from nornir_utils.plugins.functions import print_result, print_title
from modules.tasks.delete_configuration import DeleteConfiguration
from modules.tasks.eigrp_configuration import EIGRPConfiguration
from modules.tasks.interfaces_configuration import InterfacesConfiguration
from modules.tasks.linux_configuration import LinuxConfiguration
from modules.tasks.nat_configuration import NATConfiguration
from modules.tasks.ospf_configuration import OSPFConfiguration
from modules.tasks.packet_filter_configuration import PacketFilterConfiguration
from modules.tasks.static_configuration import StaticRoutingConfiguration
from modules.utility.credential_handler import CredentialHandler
from modules.utility.network_info_collector import NetworkInfoCollector
from modules.utility.network_info_exporter import NetworkInfoExporter
from modules.utility.network_info_viewer import NetworkUtilityViewer


def setup_inventory() -> Nornir:
    """
    Funkce, která umožňuje načíst veškeré informace o hostech a využívaných
        skupinách (groups). Podporuje dynamické načítání citlivých údajů (pouze
        pro citlivé údaje skupin).

    Returns:
        Nornir - nornir objekt, který obsahuje zparsované informace o hostech,
            skupinách. Dále zajišťuje multithreading funkcionalitu.
    """
    # Nornir objekt, který přeskočí zařízení, které nezvládly požadovaný (sub)
        task. více o chybě v nornir.log
    nr = InitNornir(config_file="config.yml")
```

```
    creds_handler = CredentialHandler()
    creds_handler.insert_creds(nr)
    return nr


def configure_linux_servers(nornir_devices: Nornir, task_func: callable,
    task_name: str, enable: bool = True) -> None:
    """
    Wrapper funkce, která slouží pro konfiguraci serverů založených na Linuxu.
        Funkce obaluje specifikovaný nornir úkol (task_func), který se týká
        konfigurace Linux serverů.

    Args:
        nornir_devices (Nornir): Nornir objekt, umožňující volat paralelně nornir
            úkoly (tasky) a agregovat výsledky z jednotlivých tasků pro daná zaří
            zení. Obsahuje zparsovaný inventář.
        task_func (callable): Funkce, která bude paralelně vykonávaná nornirem.
        task_name (str): Název nornir úkolu
        enable (bool): Argument, kterým lze specifikovat, jestli je nutný pro dan
            ý nornir úkol práv superuživatele (rootu). Defaultně je nastaveno na
            True (root práva).

    Returns:
        None
    """
    result = nornir_devices.run(task=task_func, name=task_name, enable=enable)
    print_result(result)


def configure_network_devices(nornir_devices: Nornir, task_func: callable,
    task_name: str, dry_run: bool) -> None:
    """
    Wrapper funkce, která slouží pro konfiguraci síťových zařízení (routerů,
        switchů). Funkce obaluje specifikovaný nornir úkol (task_func), který se
        týká konfigurace síťových prvků.

    Args:
```

```
        nornir_devices (Nornir): Nornir objekt, umožňující volat paralelně nornir
            úkoly (tasky) a agregovat výsledky z jednotlivých tasků pro daná zaří
            zení. Obsahuje zparsovaný inventář.
        task_func (callable): Funkce, která bude paralelně vykonávaná nornirem.
        task_name (str): Název nornir úkolu
        dry_run (bool): argument, který rozhoduje, jestli má být konfigurace
            provedena v testovacím režimu
        (obdržení konečných změn v konfiguraci bez jejich uložení do zařízení) -
            True. Defaultně False - uložení konečných změn.
    Returns:
        None
    """
    result = nornir_devices.run(task=task_func, name=task_name, dry_run=dry_run)
    print_result(result)


def send_command(nornir_devices: Nornir, command_string: str, task_name: str,
    enable=False) -> None:
    """
    Funkce, která slouží pro vykonání příkazu síťovými prvky pomocí knihovny
        Netmiko.

    Args:
        nornir_devices (Nornir): Nornir objekt, umožňující volat paralelně nornir
            úkoly (tasky) a agregovat výsledky z jednotlivých tasků pro daná zaří
            zení. Obsahuje zparsovaný inventář.
        command_string (str): Příkaz, který bude proveden.
        task_name (str): Název nornir úkolu
        enable (bool): Argument, kterým lze specifikovat, jestli je nutné pro
            vykonání příkazu vstoupit do privilegovaného režimu. Defaultně je
            nastaveno False (neprivilegovaný režim).

    Returns:
        None
    """
    result = nornir_devices.run(task=netmiko_send_command, command_string=
        command_string, name=task_name, enable=enable)
    print_result(result)
```

```
def main() -> None:
    """
    Hlavní funkce skriptu, která se zavolá po spuštění skriptu main.py

    Returns:
        None
    """
    #Parsování inventáře
    nornir_obj = setup_inventory()

    #Filtrování Nornir objektů
    routers = nornir_obj.filter(F(dev_type="router"))
    juniper_devices = nornir_obj.filter(F(groups__contains="juniper"))
    l3_switches = nornir_obj.filter(F(dev_type="L3_switch"))
    cisco_routers = nornir_obj.filter(F(groups__contains="cisco") & F(dev_type="
        router"))
    l3_cisco = nornir_obj.filter(F(groups__contains="cisco") & F(dev_type="
        router") | F(dev_type="L3_switch"))
    l3_devices = nornir_obj.filter(F(dev_type="router") | F(dev_type="L3_switch"
        ))
    ubuntu_servers = nornir_obj.filter(F(dev_type="ubuntu_server") | F(
        groups__contains="linux"))
    mls1 = nornir_obj.filter(F(name__contains="MLS1"))
    mls1_r3 = nornir_obj.filter(F(name__contains="MLS1") | F(name__contains="R3"
        ))

    viewer = NetworkUtilityViewer()
    exporter = NetworkInfoExporter(NetworkInfoCollector())
    ospf_config = OSPFConfiguration()
    eigrp_config = EIGRPConfiguration()
    static_routing_config = StaticRoutingConfiguration()
    interfaces_configuration = InterfacesConfiguration()
    packet_filter = PacketFilterConfiguration()
    linux_config = LinuxConfiguration()

    nat_config = NATConfiguration()
    delete_config = DeleteConfiguration()
    static_routing_config = StaticRoutingConfiguration()
```

```
#Příklady konfigurace síťových zařízení
configure_network_devices(l3_devices, interfaces_configuration.
    configure_ipv4_interfaces, "IPv4 interfaces config", dry_run=False)
configure_network_devices(l3_devices, interfaces_configuration.
    configure_ipv6_interfaces, "IPv6 interfaces config", dry_run=False)
configure_network_devices(l3_switches, interfaces_configuration.
    configure_switching_interfaces, "Switching interfaces config", dry_run=
    False)
configure_network_devices(routers, ospf_config.configure_ospf, "OSPFv2
    config", dry_run=False)
configure_network_devices(routers, ospf_config.configure_ospfv3, "OSPFv3
    config", dry_run=False)
configure_network_devices(mls1_r3, eigrp_config.configure_eigrp_ipv4, "EIGRP
     config", dry_run=False)
configure_network_devices(mls1_r3, eigrp_config.configure_eigrp_ipv6, "EIGRP
     IPV6 config", dry_run=False)
configure_network_devices(mls1, packet_filter.configure_ipv4_packet_filters,
     "IPv4 packet filter config",dry_run=False)
configure_network_devices(mls1, packet_filter.configure_ipv6_packet_filters,
     "IPv6 packet filter config",dry_run=False)

# Mazání konfigurace
# configure_network_devices(l3_devices, delete_config.delete_configuration,
    "Delete Configuration", dry_run=False)

#Sběr a výpis dat - pouze několik příkladů
l3_switches.run(task=viewer.show_vlans, json_out=False)
l3_switches.run(task=viewer.show_vlans, json_out=True)
l3_devices.run(task=viewer.show_ospf_neighbors, ipv6=True)


#Export dat
l3_devices.run(task=exporter.export_device_configuration)
l3_devices.run(task=exporter.export_packet_filter_info)
l3_devices.run(task=exporter.export_ipv4_routes)
l3_devices.run(task=exporter.export_ipv6_routes)

#Tvorba Excel reportů
```

```
    exporter.export_device_facts(l3_devices)
    exporter.export_interfaces_packet_counters(l3_devices)

    #Zasílání příkazů uvedených v inventáři (Ubuntu 18.04)
     ubuntu_servers.run(task=linux_config.send_commands, enable=True)

     #Konfigurace služby vsftpd (Ubuntu 18.04)
    ubuntu_servers.run(task=linux_config.configure_vsftpd, enable=True)


if __name__ == "__main__":
    try:
        main()
    except (NornirSubTaskError, NornirExecutionError) as err:
        print_title("Failed hosts - more in nornir.log:")
        for host in err.failed_hosts:
            host_tasks_name = []
            print(f"{Fore.RED}{host}: Nornir (sub)tasks failed: ", sep="")
            for task in err.failed_hosts[host]:
                if task.name not in host_tasks_name:
                host_tasks_name.append(task.name)
                print(f"{task.name}", sep=", ")
```

Výpis 55: Spustitelný skript main.py

```
import logging

from colorama import Fore
from nornir.core import Task
from nornir.core.exceptions import NornirSubTaskError
from nornir_jinja2.plugins.tasks import template_file
from nornir_napalm.plugins.tasks import napalm_configure
from nornir_utils.plugins.functions import print_result
from nornir_utils.plugins.tasks.data import load_yaml


class OSPFConfiguration:
    """
    Třída pro konfiguraci OSPF a OSPFv3.
    """
```

```
def configure_ospf(self, task: Task, dry_run: bool = False) -> None:
    """
    Metoda pro konfiguraci OSPFv2.

    Args:
        task (Task): Task objekt, umožňující paralelně volat a seskupovat další
            nornir úkoly (funkce).
        dry_run (bool): argument, který rozhoduje, jestli má být konfigurace
            provedena v testovacím režimu
        (obdržení konečných změn v konfiguraci bez jejich uložení do zařízení)
            - True. Defaultně False - uložení konečných změn.

    Raises:
        NornirSubTaskError: Výjimka, která nastane, pokud nastana chyba v
            nornir úkolu nebo pokud provádíte
        konfiguraci na nepodporovaných zařízeních.

    Returns:
        None
    """

    if not task.host["dev_type"] == "switch":
        data = task.run(task=load_yaml, file=f'inventory/host_vars/{task.host.
            name}.yml', name="Load host data", severity_level=logging.DEBUG)

        ospf_key = "ospf_config"

        if ospf_key in data[0].result:
            task.host[ospf_key] = data[0].result[ospf_key]

            r = task.run(task=template_file,name="OSPF Template Loading",
                template="ospf_ipv4.j2", path=f"templates/{task.host['vendor
                ']}/{task.host['dev_type']}")

            task.host["ipv4_ospf"] = r.result

            task.run(task=napalm_configure, name="Loading OSPFv2 Configuration
                on the device",replace=False,configuration=task.host["ipv4_ospf"
```

```
            ],dry_run=dry_run)
        else:
            print(f"{Fore.RED}Device {task.host.name}: No {ospf_key} key was
                found in host data.")
    else:
        print(f"{Fore.RED} Device {task.host.name}: invalid device type.")
        raise NornirSubTaskError("Invalid device type. Only routers and
            L3_switches are supported.", task)


def configure_ospfv3(self, task: Task, dry_run: bool = False) -> None:
    """
    Metoda pro konfiguraci OSPFv3.

    Args:
        task (Task): Task objekt, umožňující paralelně volat a seskupovat další
             nornir úkoly (funkce).
        dry_run (bool): argument, který rozhoduje, jestli má být konfigurace
            provedena v testovacím režimu
        (obdržení konečných změn v konfiguraci bez jejich uložení do zařízení)
            - True. Defaultně False - uložení konečných změn.

    Raises:
        NornirSubTaskError: Výjimka, která nastane, pokud nastana chyba v
            nornir úkolu nebo pokud provádíte
        konfiguraci na nepodporovaných zařízeních.

    Returns:
        None
    """

    if task.host["dev_type"] == "router":

        data = task.run(task=load_yaml, file=f'inventory/host_vars/{task.host.
            name}.yml', name="Load host data", severity_level=logging.DEBUG)
        ospfv3_key = "ospfv3_config"

        if ospfv3_key in data[0].result:
            task.host[ospfv3_key] = data[0].result[ospfv3_key]
```

```
            r = task.run(task=template_file,name="OSPF Template Loading",
                template="ospfv3.j2",path=f"templates/{task.host['vendor']}/{
                task.host['dev_type']}")

            task.host["ipv6_ospf"] = r.result

            task.run(task=napalm_configure,name="Loading OSPFv3 Configuration on
                 the device",replace=False,configuration=task.host["ipv6_ospf"],
                dry_run=dry_run)
        else:
            print(f"{Fore.RED}Device {task.host.name}: No {ospfv3_key} key was
                found in host data.")
    else:
        print(f"{Fore.RED} Device {task.host.name}: invalid device type.")
        raise NornirSubTaskError("Invalid device type. Only routers are
            supported.", task)
```

Výpis 56: Skript ospf_configuration.py

```
from datetime import datetime
from time import sleep
from typing import Dict, List
from colorama import Fore
from influxdb import InfluxDBClient
from nornir import InitNornir
from nornir.core import Nornir
from nornir_napalm.plugins.tasks import napalm_get
from modules.utility.credential_handler import CredentialHandler
from nornir_utils.plugins.functions import print_title


def setup_inventory() -> Nornir:
    """
    Funkce, která umožňuje načíst veškeré informace o hostech a využívaných
        skupinách (groups). Podporuje dynamické načítání citlivých údajů (pouze
        pro citlivé údaje skupin).

    Returns:
```

```
        Nornir - nornir objekt, který obsahuje zparsované informace o hostech,
            skupinách. Dále zajišťuje multithreading funkcionalitu.
    """
    creds_handler = CredentialHandler()
    nr = InitNornir(config_file="config.yml")
    creds_handler.insert_creds(nr)
    return nr


class DBHandler:
    """
    Třida, která slouží jako rozhraní pro práci s InfluxDB. Používána např. pro
        pravidelný zápis NAPALM dat do DB nebo pro zjištění stavu stavu jednotlvi
        ých DB.

    Attributes:
        nr_obj (Nornir): Nornir objekt, umožňující volat paralelně nornir úkoly (
            tasky) a agregovat výsledky z jednotlivých tasků pro daná zařízení.
    """

    def __init__(self):
        self._nr_obj: Nornir = setup_inventory()

    def show_db_state(self, db_conn: InfluxDBClient) -> None:
        """
        Metoda pro zjištění stavu DB (databáze, measurements, datové body).

        Args:
            db_conn (InfluxDBClient): connection objekt, který slouží jako klient
                pro připojení k InfluxDB. Dále obsahuje operace pro práci s
                InfluxDB.

        Returns:
            None
        """

        print(f"{Fore.GREEN} List of databases:")
        print(db_conn.get_list_database())
        print(f"{Fore.GREEN} List of measurements:")
```

```
    print(db_conn.get_list_measurements())

    print(f"{Fore.GREEN} List of series:")
    print(db_conn.get_list_series())

    for measurement in db_conn.get_list_measurements():
        measurement_name = str(measurement['name'])
        print(f"{Fore.GREEN} Measurement: {measurement_name}")
        query = f"SELECT * FROM {measurement_name};"
        rs = db_conn.query(query)
        for row in list(rs.get_points(measurement=measurement_name)):
            print(row)

def _get_measurement(self, napalm_key: str) -> str:
    """
    Metoda, která vrací název InfluxDB měření (measurement) dle napalm klíče
        (podle toho, co zrovna chceme zapsat do DB).

    Args:
        napalm_key (str): napalm klíč (dle použité NAPALM getter funkce)

    Returns:
        Vrací název InfluxDB měření (measurement).
    """
    if napalm_key == "environment":
        return "hw_details"
    elif napalm_key == "facts":
        return "device_facts"
    return ""

def _get_monitored_fields_values(self, napalm_key: str, host_dict: Dict) ->
    Dict:
    """
    Metoda, která vrací jednotlivé sloupce (s daty) - fields, které budou ulo
        ženy do InfluxDB.

    Args:
        napalm_key (str): napalm klíč (dle použité NAPALM getter funkce)
        host_dict (Dict): nesparsovaný Python slovník, který obsahuje data,
```

```
            které byly získány pomocí NAPALM getteru (týká se konkrétního hosta
            ).

        Returns:
            Vrací Python slovník, který vrací jednotlivé sloupce (s daty ) - fields
                . V případě prázdného host_dict nebo špatného napalm_klíče je
                vracen prázdný slovník.
        """
        fields_dict = {}
        if napalm_key == "environment" and host_dict:
            fields_dict['cpu_usage'] = host_dict['cpu'][0]['%usage']
        elif napalm_key == "facts" and host_dict:
            fields_dict['uptime'] = host_dict['uptime']
        return fields_dict

    def _write_to_db(self, host: str, measurement: str, fetch_time_utc: str,
        fields_dict: Dict, db_conn: InfluxDBClient) -> None:
        """
        Metoda, která slouží pro samotný zápis datových bodů do InfluxDB.
            Vypisuje informaci o (ne)úspěšném uložení datových bodů.

        Args:
            db_conn (InfluxDBClient): connection objekt, který slouží jako klient
                pro připojení k InfluxDB. Dále obsahuje operace pro práci s
                InfluxDB.
            measurement (str): název InfluxDB měření (measurement).
            fetch_time_utc (str): časové razítko (v UTC) - určuje kdy byla získána
                data pomocí NAPALM getterů.
            fields_dict (Dict): Python slovník, který obsahuje jednotlivé sloupce (
                s daty ) - fields.
            db_conn (InfluxDBClient): connection objekt, který slouží jako klient
                pro připojení k InfluxDB. Dále obsahuje operace pro práci s
                InfluxDB.

        Returns:
            None
        """
        is_saved = False
        if fields_dict:
```

```
            json_body = [
                {
                    "measurement": measurement,
                    "tags": {
                    "host": f"{host}"
                    },
                    "time": f"{fetch_time_utc}",
                    "fields": fields_dict
                }
            ]
            is_saved = db_conn.write_points(json_body)
        save_message = f"{Fore.GREEN}[{fetch_time_utc}] {host}: Measurement of {
            measurement} was successfuly saved." if is_saved and fields_dict else
            f"{Fore.RED}[{fetch_time_utc}] {host}: Measurement of {measurement}
            was not successfuly saved."
        print(save_message)

    def write_monitored_data(self, db_conn: InfluxDBClient) -> None:
        """
        Metoda, která slouží k pravidélnemu zápisu dat do InfluxDB. Zápis je prov
            áděň v nekonečné smyččce.

        Args:
            db_conn (InfluxDBClient): connection objekt, který slouží jako klient
                pro připojení k InfluxDB. Dále obsahuje operace pro práci s
                InfluxDB.

        Returns:
            None
        """
        while True:
            aggregated_result = self._nr_obj.run(task=napalm_get, name="Get
                env_details and device facts", getters=["environment", "facts"])
            data_fetch_time_utc = str(datetime.utcnow())
            for host in aggregated_result:
                if host not in aggregated_result.failed_hosts:
                    for key, result_dict in aggregated_result[host].result.items():
                        measurement = self._get_measurement(key)
                        fields = self._get_monitored_fields_values(key, result_dict)
```

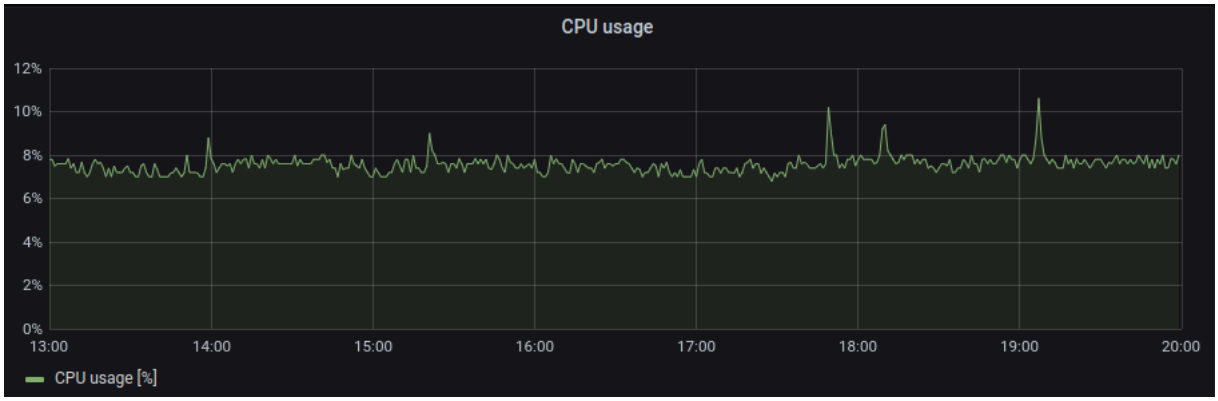
```
                self._write_to_db(host, measurement, data_fetch_time_utc,
                    fields, db_conn)
            else:
                print(f"{Fore.RED}[{data_fetch_time_utc}] {host}: Failure during
                    data collection (using NAPALM getters). Device is not probably
                     supported by used NAPALM getter.")
        sleep(10)

    def drop_db_measurements(self, db_conn: InfluxDBClient, measurements: List[
        str]) -> None:
        """
        Metoda, která slouží k mazání InfluxDB měření (včetně všech dat, která
            jsou součástí daného měření).

        Args:
            db_conn (InfluxDBClient): connection objekt, který slouží jako klient
                pro připojení k InfluxDB. Dále obsahuje operace pro práci s
                InfluxDB.
            measurements (List[str]): list, obsahující jednotliva InfluxDB měření (
                measurement).

        Returns:
            None
        """
        for measurement in measurements:
            db_conn.drop_measurement(measurement)


if __name__ == '__main__':
    db_nornir_conn = InfluxDBClient(host='10.10.10.6', port=8086, username='
        monitoring', password='monitoring', database='monitoring_nornir')
    db_ansible_conn = InfluxDBClient(host='10.10.10.6', port=8086, username='
        monitoring', password='monitoring', database='monitoring_ansible')
    db_telegraf_conn = InfluxDBClient(host='10.10.10.6', port=8086, username='
        monitoring', password='monitoring', database='monitoring_telegraf')
    db_writer = DBHandler()

    """Zápis dat do databáze monitoring_nornir- nekonečná smyčka, nutno
        zakomentovat, pokud chcete zjistit pouze stav jednotlivých databází"""
```

```
db_writer.write_monitored_data(db_nornir_conn)

#Smazat všechna měření jednotlivých databází
#db_writer.drop_db_measurements(db_nornir_conn, ['hw_details', 'device_facts
    '])
#db_writer.drop_db_measurements(db_ansible_conn, ['hw_details', '
    device_facts'])
#db_writer.drop_db_measurements(db_telegraf_conn, ['cpu', 'system'])

#Stav jednotlivých databází
#print_title("Ansible DB info")
#db_writer.show_db_state(db_ansible_conn)
#print_title("Nornir DB info ")
#db_writer.show_db_state(db_nornir_conn)
#print_title("Telegraf DB info ")
#db_writer.show_db_state(db_telegraf_conn)
```

Výpis 57: Skript db_handler.py

O Grafy - Nornir + TIG stack

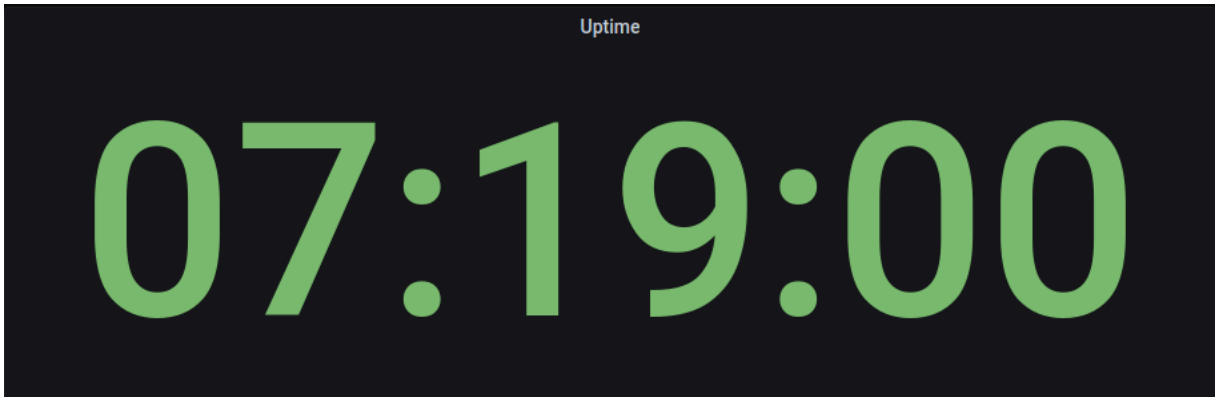
O.1 Síťové zařízení R1



Obrázek 51: Vytížení procesoru síťového zařízení R1

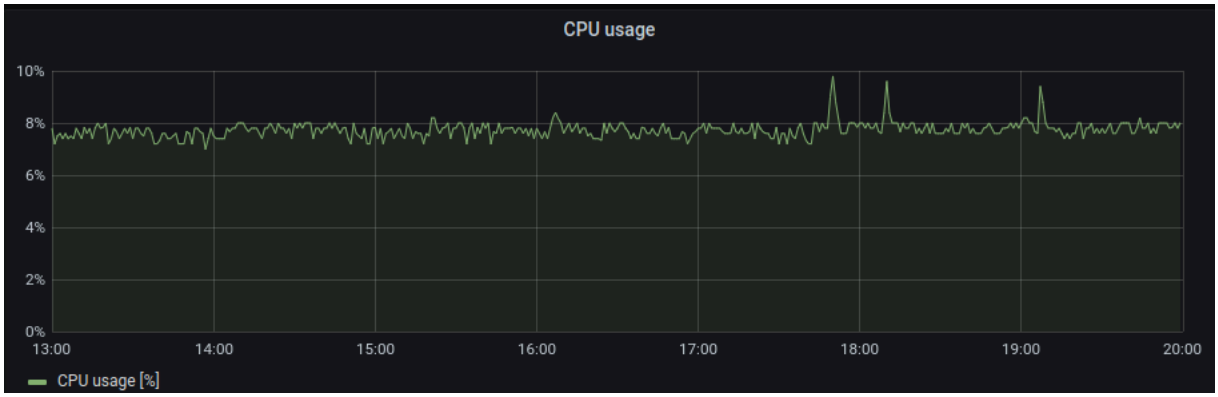
Obrázek 52: Uptime síťového zařízení R1

O.2 Síťové zařízení R2

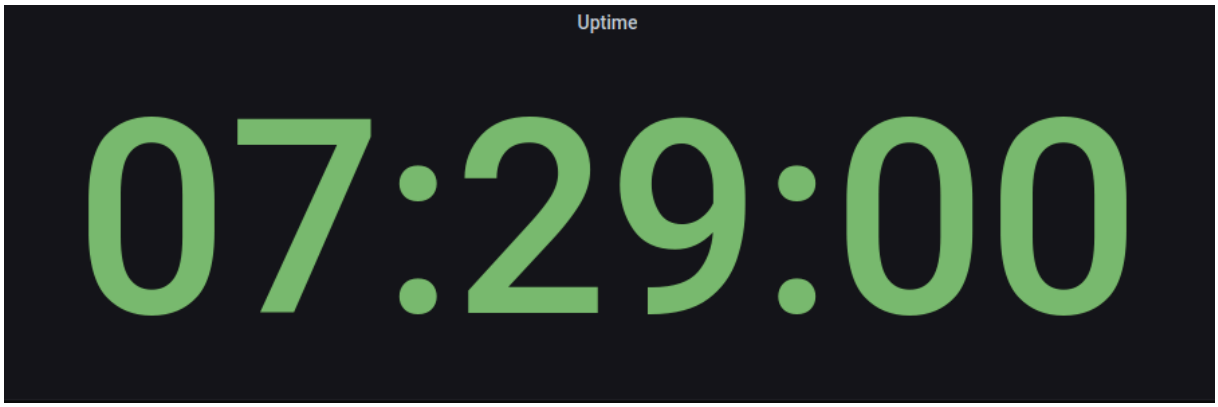


Obrázek 53: Uptime síťového zařízení R2

O.3 Síťové zařízení R3

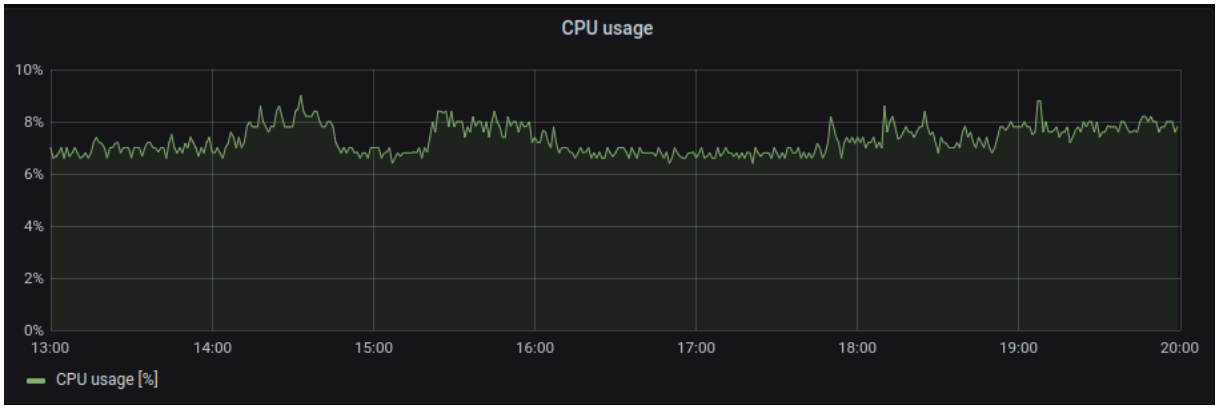


Obrázek 54: Vytížení procesoru síťového zařízení R3



Obrázek 55: Uptime síťového zařízení R3

O.4 Síťové zařízení MLS1



Obrázek 56: Vytížení procesoru síťového zařízení MLS1

Obrázek 57: Uptime síťového zařízení MLS1